Adobe® FrameMaker® 7.1

Structure Application Developer's Guide Online Manual

Adobe, the Adobe logo, Acrobat, Acrobat Reader, Adobe Type Manager, ATM, Display PostScript, Distiller, Exchange, Frame, FrameMaker, FrameViewer, InstantView, and PostScript are trademarks of Adobe Systems Incorporated. Apple, PowerBook, QuickTime, Mac, Macintosh and Power Macintosh are trademarks of Apple Computer, Inc., registered in the United States and other countries. HP-UX is a registered trademark of Hewlett-Packard Company. Microsoft, MS-DOS, Windows, and Windows NT are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Sun and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. Unix is a registered trademark and X Window System is a trademark of The Open Group. All other trademarks are property of their respective owners. © 2003 Adobe Systems Incorporated. All rights reserved.

Contents

Before You Begin - - -

Markup documents 18
FrameMaker documents 19
Multiple-file documents 19
Format rules 19
Graphics 20
Equations 20
Tables 21
Cross-references 21
Processing instructions 22
Parameter entity declarations 22

Chapter 4 *The SGML and FrameMaker Models* - - - - - 23

SGML declaration 23
SGML features with no counterparts 23
Marked sections and conditional text 23
Unsupported optional SGML features 24

Chapter 5 *The XML and FrameMaker Models* - - - - - 25

Namespace declaration 25
Rubi text 26
Element and attribute definition 26
Supported characters in element and attribute names 26
Multiple attribute lists for an element 27
Unicode and character encodings 27
Supported encodings 27
FrameMaker display encodings 28
Encoding for XML files 28

Chapter 6 *Creating a Structure Application* - - - - - 29

The development process 30
Task 1. Producing an initial EDD and DTD . . 30
Task 2. Getting sample documents 33
Task 3. Creating read/write rules 35
Task 4. Finishing your application 38
For more information 40
Pieces of a structure application 41
Application definition file 41
External DTD subset 41
SGML declaration 41
FrameMaker template 42
Read/write rules document 42
Entity catalogs 42
Documentation 42
Creating a FrameMaker template 43
Cross-reference formats 43
Variables 45
Special text flows to format generated lists and indexes 46
HTML mapping for export 46

Chapter 7 *Working with Special Files* - - - - - 49

Location of structure files 49
Application definition file 50
Editing an application definition file 51
Contents of structapps.fm 51
Defining an application 53
Providing default information 54
Specifying a document element 55
Specifying a read/write rules document 55
Specifying a DTD 56
Enabling namespaces 56

Specifying filename extensions 56
Specifying a FrameMaker template 57
Specifying an SGML declaration 57
Specifying entities 57
Specifying entities through an entity catalog . 58
Specifying the location of individual entities . . 59
Specifying names for external entity files 60
Specifying public identifiers 61
Managing CSS generation 62
Specifying a search path for external entity files 63
Specifying a search path for including files in rules documents 64
Specifying a structure API client 66
Specifying the character encoding for SGML files 66
Specifying the character encoding for XML files. 68
Limiting the length of a log file 70
Log files 71
Generating log files 71
Messages in a log file 71
Using hypertext links 72
Setting the length of a log file 72
Other special files 72

Part II Working with an EDD

Chapter 8 *Developing an Element Definition Document (EDD)* - - - - - - - - - 75

In this chapter 75
Overview of the development process 76
Creating or updating an EDD from a DTD 77
About the DTD 77
Read/write rules and the new EDD 77
Creating an EDD from a DTD 78
What happens during translation 78
Updating an EDD from a DTD 79
Log files for a translated DTD 79
Starting an EDD without using a DTD 80
Creating a new EDD 80
Exporting an Element Catalog to a new EDD 80
The Element Catalog in an EDD 81
High-level elements 81
All elements in the catalog 82
Defining preliminary settings in an EDD 88
Specifying whether to create formats automatically 89
Specifying whether to transfer HTML mapping tables 89
Setting a structure application 89
Organizing and commenting an EDD 90
Writing element definitions 91
About element tags 92
Guidelines for writing element definitions 92
Defining a container, table or footnote element . 93
Defining a Rubi group element 97
Defining an object element 99
Keyboard shortcuts for working in an EDD 102
Editing structure 102
Moving around the structure 103
Creating an Element Catalog in a template . . . 103
Importing element definitions 104
Log files for imported element definitions . . . 104
Debugging element definitions 104
Saving an EDD as a DTD for export 105

Read/write rules and the new DTD 105
Creating a DTD from an EDD 106
What happens during translation 106
SGML declarations 107
Log files for a translated EDD 107
Sample documents and EDDs 107

Chapter 9 *Structure Rules for Containers, Tables, and Footnotes* - - - - - 109

In this chapter 109
Overview of EDD structure rules 110
Writing an EDD general rule 111
Syntax of a general rule for EDD elements . 112
Restrictions on general rules for tables 114
Default general rules for EDD elements 115
Specifying validity at the highest level in a flow 116
Adding inclusions and exclusions 116
Inclusions 117
Exclusions 118
How content rules translate to markup data . . . 118
Inserting descendants automatically in containers . 119
Inserting table parts automatically in tables . . . 120
Initial structure pattern 121
Default initial structure 122
Inserting Rubi elements automatically in Rubi groups 123
Initial structure pattern 123
Debugging structure rules 123

Chapter 10 *Text Format Rules for Containers, Tables, and Footnotes* - - 125

In this chapter 125
Overview of text format rules 126
How elements inherit formatting information . . 127
The general case 128
Inheritance in a table or footnote 130
Inheritance in a document within a book . . . 131
Specifying an element paragraph format 132
Writing context-dependent format rules 132
All-contexts rules 133
Context-specific rules 134
Level rules 138
Nested format rules 140
Multiple format rules 141
Context labels 142
Defining the formatting changes in a rule 143
Paragraph formatting 143
Text range formatting 144
No additional formatting 145
Specifications for individual format properties . . 146
Basic properties 147
Font properties 150
Pagination properties 152
Numbering properties 153
Advanced properties 154
Table Cell properties 155
Asian Text Spacing properties 155
Writing first and last format rules 156
How first and last rules are applied 157
A first or last rule with an autonumber 157
Defining prefixes and suffixes 158
How prefix and suffix format rules are applied . . 158
A prefix or suffix for a text range 159
A prefix or suffix for a paragraph 159
A prefix or suffix for a sequence of paragraphs . 160
A prefix or suffix for a text range or a paragraph 161

Attributes in a prefix or suffix rule 162
When to use an autonumber, prefix or suffix, or first or last rule 162
Defining a format change list 163
Setting minimum and maximum limits on properties 165
Debugging text format rules 166

Chapter 11 *Attribute Definitions* - - - - - 169

In this chapter 169
Some uses for attributes 169
How an end user works with attributes 170
Writing attribute definitions for an element 171
Attribute name 172
Attribute type 172
Specification for a required or optional value 173
Hidden and Read-only attributes 174
List of values for Choice attributes 175
Range of values for numeric attributes 175
Default value 176
Using UniqueID and IDReference attributes . . . 176
UniqueID attributes 178
IDReference attributes 179
Using attributes to format elements 180
Using attributes to provide a prefix or suffix . . . 182

Chapter 12 *Object Format Rules* - - - - - 185

In this chapter 185
Overview of object format rules 186
Context specifications for object format rules . . 187
All-contexts rules 187
Context-specific rules 188
Setting a table format 190
Specifying a graphic content type 191
Setting a marker type 192
Setting a cross-reference format 194
Setting an equation size 194
Specifying a system variable 195
Debugging object format rules 197

Part III Translating between markup data and FrameMaker

Chapter 13 *Introduction to Translating between Markup Data and FrameMaker* - - - - - 201

In this chapter 201
What you can do with read/write rules 201
What you can do with structure API clients . . . 202
A detailed example 203
DTD fragment 203
Document instance 204
EDD fragment 205
Formatting and read/write rules 206
FrameMaker document 206
Opening XML documents 207

Chapter 14 *Read/Write Rules and Their Syntax* - - -

Chapter 17 *Translating Entities and Processing Instructions* - - - - - - - - - 247

In this chapter 247
Default translation. 248
On export to markup 248
On import to FrameMaker 251
Modifications to the default translation 257
Specifying the location of entity declarations 258
Renaming entities that become variables . . 258
Translating entity references on import and export 258
Translating entities as FrameMaker variables . 259
Translating SDATA entities as special characters in FrameMaker 260
Translating SDATA entities as FrameMaker text insets 262
Translating SDATA entities as FrameMaker reference elements 263
Translating external text entities as text insets . . 264
Translating internal text entities as text insets . . 265
Changing the structure and formatting of a text inset on import 266
Discarding external data entity references . . 267
Translating ISO public entities 267
Facilitating entry of special characters that translate as entities. 268
Creating book components from general entities 268
Discarding unknown processing instructions 268
Using entities for storing graphics or equations . 269

Chapter 18 *Translating Tables* - 271

In this chapter 271
Default translation. 272
On import to FrameMaker 272
On export to markup 275
Modifications to the default translation 275
Formatting properties for tables 276
Identifying and renaming table parts 279
Representing FrameMaker table properties as attributes in markup 280
Representing FrameMaker table properties implicitly in markup. 281
Adding format rules that use CALS attributes (CALS only) 282
Working with colspecs and spanspecs (CALS only) 283
Specifying which part of a table a row or cell occurs in 283
Specifying which column a table cell occurs in. . . 284
Omitting explicit representation of table parts. . . 285
Creating parts of a table even when those parts have no content 287
Specifying the ruling style for a table 289
Exporting table widths proportionally 290
Creating vertical straddles 290
Using a table to format an element as a boxed set of paragraphs 293
Creating tables inside other tables 295
Rotating tables on the page 295

Chapter 19 *Translating Graphics and Equations* - - - - - - - - - - - - - - - - - - - 297

In this chapter 297
Default translation 298
Supported graphic file formats 298

General import and export of graphic elements . 299
On export to markup 300
On import to FrameMaker 307
Modifications to the default translation 308
Identifying and renaming graphic and equation elements. 309
Exporting graphic and equation elements . . 309
Representing the internal structure of equations 311
Renaming markup attributes that correspond to graphic properties 311
Omitting representation of graphic properties in markup 312
Omitting optional elements and attributes from the default DTD declarations 313
Specifying the data content notation on export. . 313
Changing the name of the graphic file on export 314
Changing the file format of the graphic file on export 316
Specifying the entity name on export 318
Changing how FrameMaker writes out the size of a graphic 319

Chapter 20 *Translating Cross-References* - 321

In this chapter 321
Default translation. 321
On export to markup 322
On import to FrameMaker 323
Modifications to the default translation 324
Translating markup elements as FrameMaker cross-reference elements. 324
Renaming the markup attributes used with cross-references 325
Translating FrameMaker cross-reference elements to text in markup 326
Maintaining attribute values with FrameMaker . . 326
Translating external cross-references to and from XML 326

Chapter 21 *Translating Variables and System Variable Elements* - - - - - - 329

In this chapter 329
Default translation. 329
On export to markup 330
On import to FrameMaker 331
Modifications to the default translation 331
Renaming or changing the type of entities when translating to variables 332
Translating markup elements as system variable elements 333
Translating FrameMaker system variable elements to text in markup 333
Translating FrameMaker variables as SDATA entities. 334
Discarding FrameMaker variables 334

Chapter 22 *Translating Markers* - 335

In this chapter 335
Default translation. 335
On export to markup 336
On import to FrameMaker 336
Modifications to the default translation 337
Translating markup elements as FrameMaker marker elements 337
Writing marker text as element content instead of as an attribute 337

Using markup attributes and FrameMaker properties to identify markers 338
Discarding non-element FrameMaker markers . 339

Chapter 23 *Translating Conditional Text* - - - - - 341

In this chapter 341
Default translation 341
Condition settings 342
Conditional text 342
On export to markup 343
On import to FrameMaker 344
Modifications to the default translation 344

Chapter 24 *Processing Multiple Files as Books* - - - - - 347

In this chapter 347
Default translation 348
On import to FrameMaker 348
On export to markup 350
Modifications to the default translation 351
Using elements to identify book components on import 351
Suppressing the creation of processing instructions for a book on export 353

Chapter 25 *Read/Write Rules Summary* - - - - - 355

All Elements 355
Attributes 355
Books 356
Cross-references 356
Entities 357
Equations 357
Footnotes 358
Graphics 358
Markers 359
Processing instructions 360
Markup documents 361
Tables 361
Text 362
Text insets 362
Variables 362

Chapter 26 *Read/Write Rules Reference* - - - - - 365

anchored frame 365
attribute 367
character map 369
convert referenced graphics 372
do not include dtd 373
do not include sgml declaration 373
do not output book processing instructions . . . 373
drop 373
drop content 375
element 376
end vertical straddle 379
entity 381
entity name is 383
equation 385

export dpi is 387
export to file 389
external data entity reference 391
external dtd 392
facet 394
fm attribute 396
fm element 397
fm marker 398
fm property 399
fm variable 402
fm version 403
generate book 403
implied value is 406
include dtd 408
include sgml declaration 409
insert table part element 410
is fm attribute 414
is fm char 416
is fm cross-reference element 418
is fm element 419
is fm equation element 420
is fm footnote element 421
is fm graphic element 422
is fm marker element 423
is fm property 424
is fm property value 426
is fm reference element 430
is fm rubi element 432
is fm rubi group element 432
is fm system variable element 433
is fm table element 435
is fm table part element 436
is fm text inset 437
is fm value 439
is fm variable 441
is processing instruction 441
line break 443
marker text is 444
notation is 445
output book processing instructions 447
preserve fm element definition 448
preserve line breaks 449
processing instruction 451
proportional width resolution is 452
put element 453
reader 453
reformat as plain text 454
reformat using target document catalogs 455
retain source document formatting 456
specify size in 456
start new row 458
start vertical straddle 460
table ruling style is 461
unwrap 461
use processing instructions 463
use proportional widths 463
value 464
value is 465
write structured document 466
write structured document instance only 466
writer 467

Appendix A *Conversion Tables for Adding Structure to Documents* 471

How a conversion table works 471
Setting up a conversion table 472
Generating an initial conversion table 473
Setting up a conversion table from scratch . 474
Updating a conversion table. 474
Adding or modifying rules in a conversion table 474
About tags in a conversion table 475
Identifying a document object to wrap 476
Identifying an element to wrap 477
Identifying a sequence to wrap 478
Providing an attribute for an element 479
Using a qualifier with an element 480
Handling special cases 481
Promoting an anchored object 481
Flagging format overrides 482
Wrapping untagged formatted text 483
Nesting object elements 483
Building table structure from paragraph format tags 484
Testing and correcting a conversion table 484

Appendix B *The CALS Table Model* 487

FrameMaker properties that DO NOT have corresponding CALS attributes. 487
Element and attribute definition list declarations . . 488
Element structure 490
Attribute structure 491
Inheriting attribute values 491
Orient attribute 491
Straddling attributes 491

Appendix C *Read/Write Rules for CALS Table Model* 493

Appendix D *SGML Declaration* 497

Text of the default SGML declaration 497
SGML concrete syntax variants 499
Unsupported optional SGML features. 500

Appendix E *Character Set Mapping* 501

Appendix F *ISO Public Entities* 509

What you need to use ISO public entities. 510
Entity declaration files 511
Entity read/write rules files 511
What happens with the declarations and rules . 514

Appendix G *SGML Batch Utilities for UNIX* 517

Importing SGML documents in batch mode . . . 517
Exporting documents as SGML in batch mode . 519

Appendix H *Developing XML or SGML Publishing Applications* 521

Implementing an XML or SGML application in the FrameMaker publishing environment 521
Overview of FrameMaker Application Development. 521

Technical Steps in FrameMaker Application Development 529
Typical Application Development Scenarios . 538
Conclusions 542

Glossary 545

Index - - - - - 555

Before You Begin

This manual is for anybody who develops structured FrameMaker templates and XML or SGML applications. It is not for end users who author structured documents that use such templates and applications.

XML and SGML

FrameMaker can read and write XML (Extensible Markup Language) or SGML (Standard Generalized Markup Language) documents. XML and SGML are two types of document markup languages, and FrameMaker handles these markup languages in similar ways. However there are differences between the two, and this manual covers these differences whenever necessary.

When discussing the similarities between them, this manual refers to XML and SGML data as *markup data* or *markup documents*. Otherwise the manual refers to XML and SGML specifically to draw attention to the differences between these markup languages.

Developing structured FrameMaker templates

End users of FrameMaker can read, edit, format, and write structured documents—the structure is represented by a hierarchical tree of elements. Each structured document is based on a template that contains a catalog of element definitions. Each element definition can describe the valid contexts for an element instance, and the formatting of element instances in various contexts.

To support these end users, you create the catalog and accompanying structured template.

Developing XML and SGML applications

When the software reads markup data, FrameMaker displays the data as a formatted, structured document. When the software writes a structured FrameMaker document, the software can save the document as XML or SGML.

For the end user, this process of translation between FrameMaker documents and markup data is transparent and automatic. However, for most XML or SGML document types the translation requires an XML or SGML application to manage the translation. You develop this application to correspond with specific document types. When your end user opens a markup document with a matching document type, FrameMaker invokes the appropriate structure application. If there is no match for a document type, the user can choose the application to use, or open the markup document with no structure application. A structure application primarily consists of:

- A structured template
- Read/Write rules (described in this manual)
- An XML and SGML API client (if necessary) developed with the Frame Developer's Kit (FDK)

Prerequisites

The following topics, which are outside the scope of this manual, are important for you to understand before you try to create a structured template or structure application:

- Structured document authoring in FrameMaker
- XML or SGML concepts and syntax, including how to work with a *document type declaration*
- FrameMaker end-user concepts and command syntax
- FrameMaker template design

In creating some XML or SGML applications, you may also need to understand the following:

- C programming
- FDK API usage

If your application requires only the special rules described in this manual to modify the default behavior of FrameMaker, you do not need programming skills. However, if you need to create an XML and SGML API client to modify this behavior further, you need to program the client in C, using the FDK. This manual does not discuss the creation of XML and SGML API clients. For this information, see the *Structure Import/Export API Programmer's Guide*.

Using FrameMaker documentation

FrameMaker comes with a complete set of end-user and developer documentation with which you should be familiar. If you use the Frame Developer's Kit in creating your structure application, you'll also need to be familiar with the FDK documentation set.

Using this manual

This manual is divided into three major parts and a series of appendixes. If you're creating a structure application, you'll find information you need in all three parts. If, however, you're not working with markup data but are creating a structured template, you'll need only Part II. The parts are as follows:

- Part I, "Developing a FrameMaker structure application"

  Part I is for developers of XML or SGML structure applications. This section contains
  - introductory information
  - An overview of the steps in creating a structure application

- A comparison of FrameMaker concepts and XML and SGML
- Details of assembling the pieces of an application into a whole

- Part II, “Working with an EDD”

  Part II is for developers of a FrameMaker structured template. It contains information on how you use an element definition document (EDD) to define elements and determine their formatting for your documents. You use this part in conjunction with chapters in the FrameMaker user’s manual that describe other aspects of template creation.

- Part III, “Translating between markup data and FrameMaker”

  Part III is for developers of structure applications. FrameMaker’s default translation between markup documents and FrameMaker documents follows a model. This part describes the model and the rules you use to modify the default translation.

- Part III, “Translating between markup data and FrameMaker”

  Part IV is for developers of XML applications. FrameMaker’s default translation between XML documents and FrameMaker documents follows a model. This part describes the model and the rules you use to modify the default translation.

- Appendixes

  The appendixes include information such as how to add structure to unstructured FrameMaker documents and how to work with ISO public entities. There is also a glossary of terms.

If you’re creating an XML or SGML application, we encourage you to read the first three chapters of Part I before reading any of the rest of the manual. For your purposes, these chapters are a prerequisite to the rest of the manual.

Each of the parts of this manual begins a short introduction in which you can find information about specific chapters in that part.

Using other FrameMaker documentation

The *FrameMaker User Guide* makes up the primary end-user documentation for this product. It explains how to use the FrameMaker authoring environment for both structured and unstructured documents. It also explains how to create templates for your documents.

In creating a structured template, you can refer to this manual for information on how your end user interacts with the product and how to create a formatted template.

Using FDK manuals

If you create an XML and SGML API client for your XML or SGML application, you’ll need to be familiar with the FDK. FDK documentation is written for developers with C programming experience.

- *FDK Programmer’s Guide* is your manual for understanding FDK basics. This manual describes how to use the FDK to enhance the functionality of FrameMaker and describes

how to use the FDK to work with structured documents. To make advanced modifications to the software's default translation behavior, refer to the *Structure Import/Export API Programmer's Guide*.)

- *FDK Programmer's Reference* is a reference for the functions and objects described in the *FDK Programmer's Guide*.
- *Structure Import/Export API Programmer's Guide* explains how to use the FDK to make advanced modifications to the software's default behavior for translation between markup documents and FrameMaker documents. This manual contains both descriptive and reference information.

For information on other FDK manuals, see "Using Frame Developer Tools" in the *FDK Programmer's Guide*.

Part I Developing a FrameMaker structure application

Part I provides basic information for developing SGML applications, manual including:

- Chapter 2, “Structure Application Basics”

  Describes situations that require a structure application. Also contains a high-level description of application creation.

- Chapter 3, “A General Comparison of Markup and FrameMaker Documents”

  Compares relevant SGML and FrameMaker concepts. You should read this chapter even if you are already familiar with both SGML and FrameMaker, since translation between the two is its own distinct topic. The chapter deals with counterpart constructs in the two representations, and also with constructs in one that have no real counterpart in the other.

- Chapter 6, “Creating a Structure Application”

  Describes typical application creation workflow. Also discusses the types of files used in your final SGML application.

- Chapter 7, “Working with Special Files”

  Tells where to find special files used by FrameMaker with XML and SGML documents. Also explains creation of the file that defines the pieces of your structure application.

1 What's New in FrameMaker

This chapter describes new additions to FrameMaker that pertain to the development and use of structured applications.

What's new in FrameMaker 7.1

FrameMaker 7.1 introduces several changes in import and export of XML documents.

> **Important:** None of these changes affect import or export of SGML documents.

The following sections briefly describe the new FrameMaker 7.1 functionality and provide links to more complete descriptions.

Conditional text

FrameMaker 7.0 exported shown conditional text with no indication in the resulting XML that the content was conditional. It never exported hidden conditional text. FrameMaker 7.1 allows export of both shown and hidden conditional text. Furthermore, it optionally exports processing instructions that define condition settings and identify conditional material. It interprets these processing instructions when importing an XML document and thus supports round-tripping of conditional text. See Chapter 23, "Translating Conditional Text."

External cross-references

FrameMaker 7.0 used XML elements to export internal cross-references (those in which the cross-reference source is in the same FrameMaker document as the cross-reference itself). When it exported a book, it also generated XML elements for cross-references from one book component to another component of the same book. It converted all other external cross-references to text.

FrameMaker 7.1 exports all cross-references as XML elements. It uses a new `srcfile` attribute to identify the source file of external cross-references. You can choose whether the FrameMaker or XML form of the source file should be named. On import, FrameMaker reads the `srcfile` attribute to create external cross-references. See Chapter 20, "Translating Cross-References."

Preserving the XML extension

When FrameMaker 7.0 opened an XML document, it created a new FrameMaker document without preserving the fact that the document originated as XML. It changed the file extension to `.fm` and used this modified filename to label the resulting document window.

FrameMaker 7.1 preserves the original file name. If you save an imported XML document, the document is saved as XML rather than as a FrameMaker document. Furthermore, FrameMaker locks an open XML document so that no other user can change it, with FrameMaker or with another application, until it is closed. See "Opening XML documents" on page 207 for more details.

Priority of variable definitions

FrameMaker imports some XML entity references as user variables. If the user variable is not defined in the import template, FrameMaker defines it, using the entity definition (text in quotation marks) in the XML entity declaration as the FrameMaker variable definition. FrameMaker 7.0 and FrameMaker 7.1 differ in their treatment of entity references when the corresponding variable is predefined in the template. FrameMaker 7.0 gave priority to the definition in the template. If the definition in the template differed from that in the XML document, it used the definition in the template. FrameMaker 7.1 gives priority to the definition in the XML entity declaration. FrameMaker 7.1 therefore supports round-tripping of such entity declarations. See "Internal text entities" on page 251.

What's new in FrameMaker 7.0

FrameMaker 7.0 introduces full support for the import and export of XML markup. As a result, many things have changed. Most notably, FrameMaker now supports the import of XML with UNICODE text. You can edit the imported documents, and when you save them as XML again FrameMaker retains the font encoding.

The following sections briefly describe the changes in FrameMaker, and provide links to the parts of this manual that describe these changes in detail.

Path and filename changes

Earlier versions of FrameMaker included many path and filenames that included the letters "SGML" to identify them as specific to FrameMaker+SGML. With version 7.0 structured FrameMaker documents can be saved as SGML or XML—many of these path and filenames have changed. Also, the structure of the FrameMaker installation is slightly changed to separate files specific to SGML and XML into separate branches. Among these changes:

- What used to be the `sgml` branch of the installation is now found underneath the `structure` directory—there you can find *`$installdir`*`/structure/sgml` and *`$installdir`*`/structure/xml` which contain separate branches for files that pertain to SGML and XML
- The file `sgmlapps.fm` is renamed to `structapps.fm`—you can find this file in *`$installdir`*`/structure/structlapps.fm`

Application development

Because of the differences between SGML and XML, you must specify which structure applications are for SGML and which are for XML. For XML structure applications you can specify how an application manages:

- CSS files
- Namespaces
- Character encoding and Unicode support in XML

For more information, see Chapter 7, "Working with Special Files."

Read/write rules

Because XML is case-sensitive, any references to XML keywords in your read/write rules must match the case of these keywords. Generally, these keywords are in uppercase letters.

The names of the following read/write rules have changed:

From:	To:
fmsgml version	fm version
write sgml document	write structured document
write sgml document instance only	write structured document instance only

Namespaces

FrameMaker 7.0 supports XML namespace declarations. For more information, see "Namespace declaration" on page 25

CSS support

FrameMaker does not use CSS data when importing XML, but it does retain the CSS specification in an XML document on export. You can also generate a CSS from the formatting information that is stored in an EDD. See Chapter 15, "Saving EDD Formatting Information as a CSS Stylesheet."

Exporting variables to markup

FrameMaker exports non-element variables as entities. Depending on the variable, some of these entities can be SDATA or CDATA. XML doesn't support SDATA or CDATA entities—you have two options for handling such non-element variables:

- By default, FrameMaker writes the variables out as text
- You can use read/write rules to map the variables to empty elements

For variables that are elements, FrameMaker exports them as empty elements in markup.

For more information, see Chapter 21, "Translating Variables and System Variable Elements."

New starter kits

FrameMaker 7.0 includes a new starter kit for XHTML, and an updated kit for DocBook 4.1 (SGML) and 4.1.2 (XML). For more information, see the starter kit manuals in the online manuals directory.

2 Structure Application Basics

This chapter provides an introduction to structure application development using FrameMaker. The chapter examines some of the reasons you might be using markup data and FrameMaker together and explains how those reasons affect the structure application you develop. The chapter also provides high-level information about creating that application.

Where to begin

If your end users do not need markup, but use structured documents only because they want the benefits inherent in a structured authoring environment, you only need to provide them an *element definition document (EDD)* and a structured template. In this case, the material you need is contained in Part II, "Working with an EDD."

If your end users need to read or write markup documents, you will probably need to develop a structure application to modify the default translation between markup data and FrameMaker documents. In that case, this entire manual is of use to you. The rest of Part I provides general information on creating a structure application. Part III, "Translating between markup data and FrameMaker," provides details of the default translation between markup data and FrameMaker and explains how you can change this default behavior with structure read/write rules.

Structure application scenarios

The specific features of your structure application will largely depend on how your organization intends to use it.

Translating in one or two directions?

You can write structure applications that translate documents in one direction—from XML or SGML to FrameMaker or from FrameMaker to XML or SGML. Or your application can translate in both directions, enabling information to make round trips between markup data and FrameMaker.

If your end users only translate the data in one direction, you can limit your structure application to only manage the translation in that direction. One-direction translations are:

- Reading markup data into FrameMaker—the structure application only needs to import specific doctypes into FrameMaker
- Writing FrameMaker documents as markup data—the structure application only needs to export FrameMaker documents as XML or SGML of a specific doctype

For example, your company may have a large SGML document database of parts information from which it periodically needs to produce catalogs. You may want to deliver the information in FrameMaker to take advantage of its formatting capabilities while you continue to create and store the information only in SGML. In this case, end users manipulate the catalog document in FrameMaker, but not the source SGML database. Because of this, you set up an application to translate from SGML to FrameMaker but don't worry about the reverse.

Or assume your company needs to publish documents in print, PDF, and XML. The authors can create the documentation in FrameMaker, paying attention to pagination and other formatting issues as they go. When they have a final version of the documents they can save them as PDF and XML, and the online documentation will be made from the same source as the printed documentation.

Or perhaps your end users work with FrameMaker to create their documents, but they collaborate with other writers who use a different authoring environment for XML documents. In this situation your application needs both to read XML documents into FrameMaker and to write FrameMaker to XML, so that your end users can collaborate effectively.

Can you simplify when translating in only one direction?

If your application only needs to translate in one direction, then you might be able to simplify some of the information you present. For instance, if your end users have existing FrameMaker documents to deliver in markup, they developed documents using the full power of FrameMaker. They may well have used such FrameMaker features as markers to facilitate document creation. But some of these features may not be relevant in markup data, so you can choose to omit them from the markup document. If you won't be re-importing the markup data back into FrameMaker, the resultant loss of information is not important. Therefore, you don't have to retain the use of those features in your structure application.

Do you have an existing DTD or EDD?

Another factor influencing the design of your application is the degree to which the document structure has already been defined for the markup data and the FrameMaker template. Is your starting point an established EDD or DTD—for instance, one of the Department of Defense *CALS* DTDs? Will you be provided with both an established EDD and an existing body of documents that use that EDD? Or has neither definition document as yet been written?

It is common to start application development with a DTD. In that case, your structure application must also address the issues of translating the DTD to an EDD. It is also possible that you begin with an existing EDD, and must translate the EDD into a DTD.

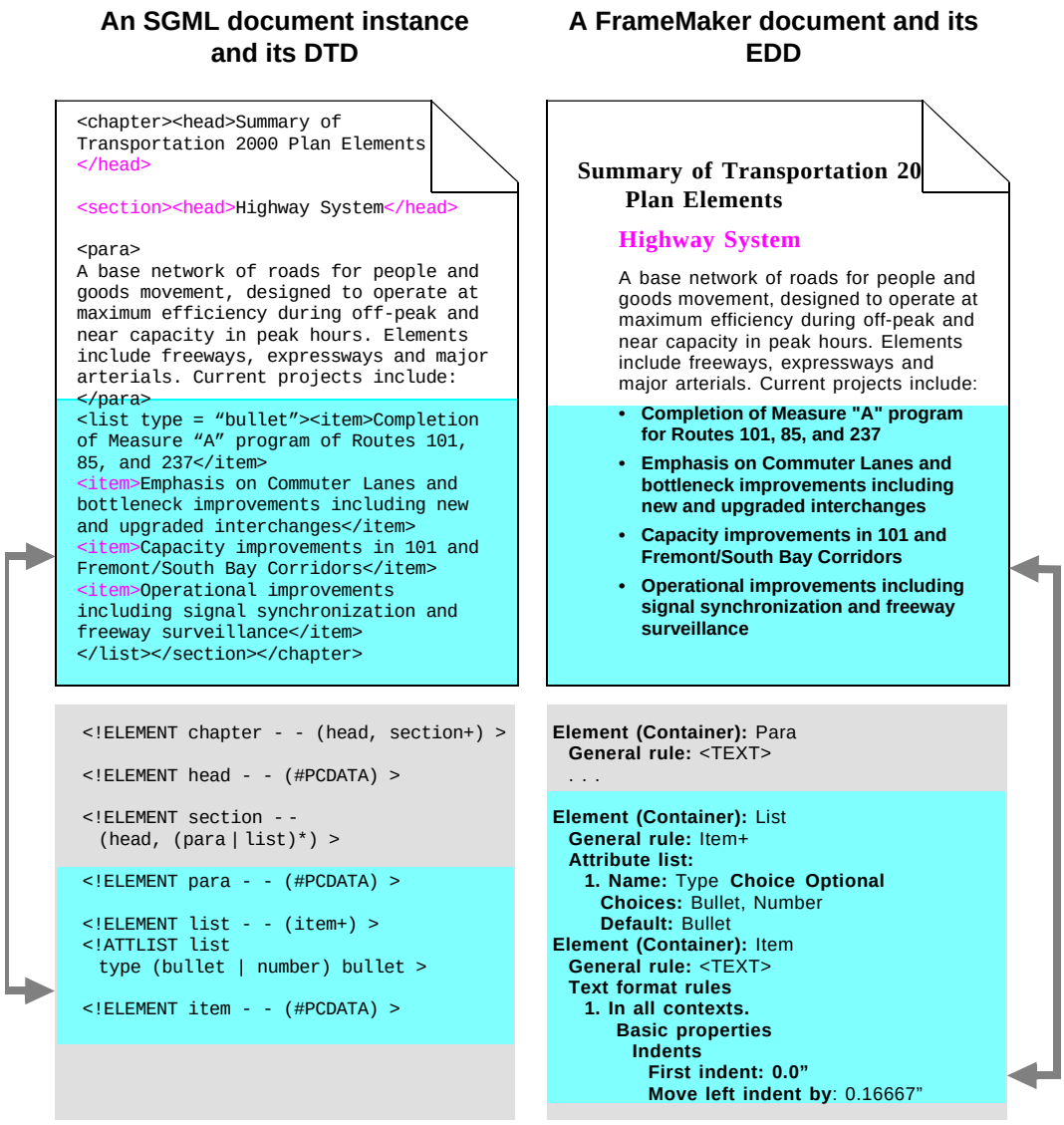
For example, assume your end users already have a library of documents based on a particular DTD. In that case, you'll be less inclined to make changes to the DTD that require

modifying those documents. If you start without a model on either side, however, you have the freedom to design the EDD and the DTD with its counterpart in mind. If you have this freedom, the application design process may be easier.

Structure application development

Markup data and structure in FrameMaker have many similarities. To describe the hierarchical nature of documents, both use special methods to define the elements that can occur in a *document instance*, and the relationships among those elements. In markup, these elements are defined in a DTD, and in FrameMaker they are defined in an EDD.

Within particular document instances, both use descriptive tags to identify each instance of a structural element. The tags in the document content correlate to the element definitions found in the DTD or the EDD, as shown in the following illustration. The top of the illustration shows pages of SGML and FrameMaker documents, while the bottom shows a DTD and EDD, respectively. The highlighted portion of the SGML document corresponds to the highlighted definition for `<list>` in the corresponding DTD. The FrameMaker page and EDD illustrate a similar correspondence.



Despite the similarities in their basic approach to the creation of structured documents, XML and SGML differ from FrameMaker in their methods of representing document content, and in the aspects of documents they represent. In this respect, markup and FrameMaker are like ordinary human languages that evolve from different cultures—they differ not only in the particular words they use, but in their rules for putting words together, and even in the thoughts that can be easily expressed with them. For documents to move between FrameMaker and markup data, the software needs instructions to help it translate between the two languages.

The starting point: an EDD or DTD

Your end users need a starting point that describes the structure of documents they'll create or edit. You provide this starting point either with a DTD or a FrameMaker EDD.

If you start with a DTD, FrameMaker can create portions of a corresponding EDD for you. Similarly, if you start with an EDD, FrameMaker can create a DTD for you. If a starting point has not yet been established, you can create your own DTD or EDD. In this situation, we recommend you to start by using FrameMaker's tools for creating an EDD (described in Part II of this manual) and then create a DTD from that EDD.

Translation between DTDs and EDDs

FrameMaker can automatically translate a DTD or EDD. If you start with a DTD, the software creates a default version of the corresponding EDD. Similarly, if you start with an EDD, FrameMaker creates a default version of the DTD.

Once your application is complete, your end users use standard commands to save individual FrameMaker documents as markup documents or to open individual markup documents in FrameMaker. In either case, the structure application works transparently with FrameMaker, making these automatic translations possible.

Formatting information in FrameMaker

You create a structured template to provide appropriate formatting for your documents. Properly speaking, a DTD does not provide any formatting information. With XML, formatting information is provided via CSS or XSL. When reading XML, FrameMaker does not use any of this formatting information.

When reading markup data, FrameMaker combines any formatting that is defined in the document template with any formatting rules that are specified in the EDD. To vary the formatting of your documents, you may create more than one structured template for the same DTD. For example, your end users can use the same markup data for different purposes, such as a fully designed catalog and a brief parts list. They may want the same data formatted differently in these two situations.

Changing the default translation

The structure of markup data varies widely, and that can affect the way it maps to FrameMaker document objects. As a result, much of your job in creating a structure application is to change the default translation FrameMaker uses. You do so by providing information the software needs to recognize and process particular constructs. If FrameMaker automatically translates all the components of a document just as you want, you don't need to provide this extra information.

One of the differences between markup data and FrameMaker is that has explicit representations for items such as tables or graphics. With XML or SGML, an individual DTD defines a arbitrary representations for tables and graphics. For this reason, the default translation in FrameMaker assume certain DTD representations of these items. For

example, by default FrameMaker assumes a CALS model for tables. These assumptions may or may not match your actual DTD. If they don't match, you need to develop an application that modifies the default translation accordingly.

How you modify the translation

To modify how FrameMaker translates documents, you start by using special *read/write rules*. In many cases these rules are sufficient to successfully translate a given doctype. A significant part of your effort in creating a structure application consists of specifying these rules.

You specify read/write rules in a special rules document. Most rules are relevant to all import or export functions, but a small number apply only to some. For example, there is a rule to rename an element on import and export. There is also a rule to tell the software how to treat FrameMaker non-element markers on export, but this rule doesn't apply to import or to creation of a DTD.

In some situations the representation you want in your DTD and your EDD may be radically different from what FrameMaker does by default. If so, rules might not be adequate to correctly translate your documents. For example, the FrameMaker model for tables assumes that a table is described in row-major order. But if your DTD describes tables in column-major order, then the FrameMaker read/write rules won't be able to translate between the markup data and FrameMaker representations. In other situations, your XML or SGML document may contain *processing instructions* that need to be interpreted but are unknown to FrameMaker. In such situations, you can customize FrameMaker via the FDK.

The FDK allows you to modify FrameMaker's import and export of individual markup and FrameMaker documents. You cannot use it to modify the creation of an EDD or a DTD.

What your end users do

Once you've created a structure application, your end users use that application to open markup documents in FrameMaker and to save FrameMaker documents as markup data. For this purpose, they'll work in a FrameMaker environment that includes your structure application. Depending on your application, the fact that files are stored as markup or as FrameMaker documents can be invisible to your end users. For example, the standard Open command opens either an XML or a FrameMaker document.

3 A General Comparison of Markup and FrameMaker Documents

Markup data and FrameMaker documents use models for structured documents that are sometimes similar and sometimes substantially different. This chapter describes the similarities and differences that you should understand before you can create a structure application.

Structure descriptions

In markup data, elements are defined in *element declarations* and *attribute definition list declarations* within a *document type definition (DTD).* In FrameMaker elements are defined in *element definitions* within an *element definition document (EDD)* and attributes are defined as part of an element.

FrameMaker EDDs

In FrameMaker, the EDD in which you create element definitions is a separate document. After creating the file of element definitions, you import the definitions into a template. A *template* is a FrameMaker document that stores information about a set of documents. This information includes element definitions and many other details such as paragraph formats, table formats, page layouts, and variable definitions.

The process of importing the EDD into a template stores the element definitions in the template's *Element Catalog*. After the EDD designer has imported the EDD, it is no longer needed as a separate document. Typically, the EDD designer retains the separate EDD document for maintenance. End users such as writers or editors, however, don't need direct access to the EDD. Instead, they work exclusively with the template file.

For information on creating EDDs, see Part II, "Working with an EDD."

XML and SGML DTDs

For markup data, the process of providing declarations for a document is somewhat different. The DTD designer can create the declarations in a separate file. However, there is no step of transforming a DTD for use by a particular document. The form of the DTD remains constant.

In addition, a DTD is only concerned with the syntax of a document—that is, with legal ways to put together the pieces of a document, regardless of the intended purpose of the pieces. Markup has nothing to say about the semantics of the process, that is, the meaning of those pieces.

Before its document instance, a markup document always includes either all the declarations or a reference to them. Rather than requiring that all of the declarations in a DTD be included in each markup document, the XML and SGML standards allow for the declarations to be stored separately and referenced from the document. The document type declaration in such a document includes the set of declarations stored separately by referring to them as an external entity.

A typical document type declaration has the form:

```
<!DOCTYPE name extid [ internal_declarations ] >
```

For XML, a document type declaration can have the form:

```
<!DOCTYPE name extid URL [ internal_declarations ] >
```

In these examples, *name* is the document type name and *extid* is the external identifier of an entity that consists of a list of declarations. For XML, *URL* refers to the location of the external declarations on the internet.

> **Important:** When reading and writing XML data, FrameMaker ignores the URL statement in the document type declaration. In FrameMaker, the structure application can include a specific DTD, or it can use a map to resolve the public identifier for the DTD on your system.

The declarations in the external entity are treated as though they appeared at the end of *internal_declarations*. The declarations that actually appear in *internal_declarations* are read before the ones in the external entity. Together, the declarations in the external entity and in *internal_declarations* are the *document type declaration subset* of the DTD.

There is an informal practice in the SGML community of using the term *external DTD subset* to refer to this external entity and using the term *internal DTD subset* to refer to the declarations shown here as *internal_declarations*.

In most places in this manual that use the term DTD, it can refer to either a complete DTD or to an external DTD subset. In the few places where the distinction matters, the manual clarifies which one is meant.

Elements

In both markup data and in FrameMaker, the basic building blocks of documents are *elements*. Elements hold pieces of a document's content (its text, graphics, and so on) and together make up the document's structure. FrameMaker distinguishes among several specific element types, corresponding to special FrameMaker document objects such as tables or cross-references. Markup does not make such a distinction.

FrameMaker element types

A large proportion of the elements in a structured FrameMaker document are defined to include text, child elements, or both. FrameMaker has several additional element types that represent constructs in a document for which the software has special authoring tools. These element types are for

- footnotes
- cross-references
- markers
- system variables
- equations
- graphics
- tables
- table parts (such as table footings)

If the element does not correspond to one of these constructs for which FrameMaker has special tools, the element is tagged as a *container* in the EDD.

FrameMaker provides flexibility in handling markers and system variables. As noted above, you can define elements that correspond to them. Alternatively, you can allow users to create *non-element* markers or system variables directly in an element that can contain text. In this case, you do not create a corresponding element. You choose the appropriate representation depending on your end-user's needs. For example, if your EDD includes a definition for a marker element, then when an end user inserts an element of this type in a document, FrameMaker uses the marker element's format rules to insert a marker of the correct type. When an end user inserts a non-element marker, on the other hand, the user must explicitly specify the marker type.

For more information on element types, see Chapter 8, "Developing an Element Definition Document (EDD)." For information on translating between FrameMaker and markup elements of various types, see Part III, "Translating between markup data and FrameMaker."

XML and SGML elements

Markup data provides even more flexibility than does FrameMaker. Markup doesn't dictate the purpose of elements of any type. A construct that is represented in FrameMaker by a marker or system variable element can be represented by an element with a declared content of empty in markup. It can also be represented by a completely different element structure.

Element declarations and definitions

Each element definition or declaration in both markup data and in FrameMaker contains the element's name and information about appropriate content for that element. FrameMaker element definitions also contain appropriate formatting for the content. In markup data, an element's name is its *generic identifier*; in FrameMaker, the name is its *element tag*. The information about content is in the *declared content* or in the *content model* of a markup element declaration and in the *general rule* of a FrameMaker element definition. Formatting information is specified via the *format rules* of a FrameMaker element definition.

Element content

The content model or declared content of a markup element and the element tag and general rule of a FrameMaker element specify in precise terms what an element can contain. In FrameMaker, the part of the content rule that specifies the basic element content is the *general rule*.

A FrameMaker element's general rule can be the single specifier `<EMPTY>` to indicate no content, or a combination of text and child elements. Some of the special element types, such as cross-references, do not have a separate content rule, since an element of that type always has the same content (a single cross-reference in this case). A markup element's content model can include either the reserved name `ANY` ("anything") or a combination of child elements. In some element definitions in both markup and FrameMaker, the content rule also specifies what content is required or optional and the required order of the content.

Inclusions and exclusions

> **XML:** The XML specification doesn't allow inclusions or exclusions.

Definitions and declarations can specify elements with constraints other than their general rule. An element that is an *inclusion* in another element can occur anywhere within that element, without the general rule explicitly listing the included element. An element that is an *exclusion* to another element cannot occur anywhere within that element, even if the content rule implies that the excluded element is allowed.

For example, you can define an element representing a chapter as a heading followed by a combination of paragraphs and lists, each of which has its own structure. If you want to allow footnotes anywhere in a chapter, you could specify the element representing a footnote element as an inclusion in a chapter element. You then would not need to explicitly mention a footnote element in the definitions of paragraphs or lists, but would be able to include footnotes in those elements.

Attributes

FrameMaker and markup both provide *attributes* to supply additional information about an element. For example, the DTD designer for a manual could use an attribute called `version` for its `book` element to allow the user to specify a book's revision status.

In FrameMaker, the attributes for an element are a part of the definition of the element itself. In XML or SGML, the attributes for an element occur separately in an *attribute definition list declaration (ATTLIST)* in the DTD.

By default, FrameMaker translates most attributes in markup data as attributes of the same name in the FrameMaker document. However, you may decide to supply rules to change this behavior. Some attributes in markup data represent information best represented by other means in FrameMaker. For example, you can write a rule to specify that an attribute corresponds to a particular property, such as the number of columns in a table, instead of to a FrameMaker attribute.

For information on defining attributes in an EDD, see Chapter 11, "Attribute Definitions," in this manual. For information on translating between FrameMaker and markup attributes, see Chapter 16, "Translating Elements and Their Attributes."

Entities

In markup, an *entity* is a collection of characters you reference as a unit. Entities are used for many purposes. You can use an entity as

- shorthand for a frequently used phrase
- a placeholder for an external file containing a graphic in some special format
- a way to include multiple physical files in the same document.

FrameMaker provides several features for handling situations for which you use entities in markup data.

Entities can be classified in various ways. For example, they can be grouped into general or parameter entities or into internal or external entities. *General entities* usually occur inside a document instance. *Parameter entities* usually occur inside a DTD. *Internal entities* have replacement text specified directly in an entity declaration. *External entities* are stored in a separate storage object (such as a data file) often identified in the entity declaration by a system identifier, public identifier, or both.

While FrameMaker doesn't have a single construct called an entity, it uses various mechanisms to provide the functionality of many XML and SGML entities. Entity-like features include the following:

Text substitution You can represent frequently repeated sequences of characters as general entities in markup, and as variables or text insets in FrameMaker.

File management You can break large documents across multiple files and manage those files with a single document containing general entity references in markup, and with a book file in FrameMaker.

Graphics In markup data you often store graphics in separate files, then include them in the document with general entity name attributes. In FrameMaker, you can store graphics and equations externally or internally.

Special characters SGML doesn't allow you to enter certain characters or sequences of characters directly as data characters in an SGML document. This can happen, for example, if the character is not in the character set of the SGML document. In FrameMaker these might be either variables or characters in a particular font.

> **XML:** The XML specification allows the use of UNICODE text, which makes this use of entities largely unnecessary. The specification also identifies predefined character entities which FrameMaker can translate by default.

Markup In SGML, entities may contain actual markup. Because FrameMaker is a WYSIWYG tool, it has no concept of markup as such.

For information on creating variables, text insets, and books, see the *FrameMaker User Guide*.

For information on translating entities to various FrameMaker constructs, see Chapter 17, "Translating Entities and Processing Instructions," Chapter 19, "Translating Graphics and Equations,"Chapter 21, "Translating Variables and System Variable Elements." and Chapter 24, "Processing Multiple Files as Books."

Documents

The XML and SGML specifications use the term *document* differently than it is used in FrameMaker. A markup document has a particular set of parts in a particular order and can be spread across multiple physical files. A FrameMaker document is simply a single file in FrameMaker format.

Markup documents

According to the XML and SGML standards, an XML or SGML document contains an *SGML declaration* (SGML-only), a *prolog*, and a *document instance set*. (For more information about the SGML declaration, see Chapter 4, "The SGML and FrameMaker Models.")

The prolog and document instance set allowed with FrameMaker have the simplest form defined in the standard. For a document used with FrameMaker, its prolog can contain comments, processing instructions, and exactly one DTD. Its document instance set includes exactly one *document instance*. A document instance is a particular collection of data and markup such as a memo or book—what most users informally think of as a document.

When you open the file or files comprising an XML or SGML document, you can clearly see the parts of the document corresponding to the SGML declaration, DTD, and document instance. Frequently, the bulk of the DTD actually resides in a separate file as an external DTD subset and is referenced in the document.

FrameMaker documents

Since FrameMaker is a WYSIWYG tool, a FrameMaker document is organized differently than a markup document. A FrameMaker document contains information specified in the template from which it was created, along with the data of the document. The template information is stored in various *catalogs*, such as the *Element Catalog* and the *Paragraph Catalog*, and in special nonprinting master and reference pages. Rather than having explicit markup that appears in the document, it uses commands for adding structure and formatting that take effect immediately.

Multiple-file documents

Frequently, your end user wants to divide the text for a document into multiple physical files. For example, a book may have a separate file for each chapter. Both markup documents and FrameMaker allow a single conceptual document to be divided into multiple physical files.

FrameMaker provides the *book* mechanism for this purpose. A book file contains a list of files that together form a complete work. Each file in the book is a complete FrameMaker document and can stand on its own.

In markup, you can use text entities for the same purpose—you can have a single file with references to text entities, each of which contains a portion of the document. In markup, each text entity isn't a complete document. That is, each entity doesn't have its own DTD, and document instance. Instead, the text entities are part of the document instance of a single markup document.

For information on creating FrameMaker books, see the FrameMaker and FrameMaker user guides. For information on creating books from text entities, see Chapter 24, "Processing Multiple Files as Books."

Format rules

Markup has no standard mechanism for representing the formatting of a document, although some DTDs use attributes for some formatting information. XML uses CSS and XSL to represent formatting.

FrameMaker provides format rules that allow an EDD to store context-sensitive formatting information. When you read an XML document into FrameMaker, the formatting specified in the CSS or XSL file has no effect. Instead, FrameMaker uses the formatting specified in the EDD.

FrameMaker also uses the EDD format rules when you edit a document. As you insert an element with format rules, FrameMaker applies the appropriate format to the element's content on the basis of surrounding structure and attribute values. That is, FrameMaker can format the same element in different ways, in different contexts in a document. In addition, an end user can override formats for any portion of a document.

FrameMaker format rules can be defined hierarchically. For example, you can say that the font family and size for `Section` elements are Times 12pt and for `MyTab` table elements they are Helvetica 12pt. Later, you can say that the `Fnote` footnote element is 9pt. Since you did not specify the font family for `Fnote`, it is Times if it occurs in a `Section` element, but Helvetica if it occurs in a `MyTab` element.

For information on creating format rules in an EDD, see Chapter 10, "Text Format Rules for Containers, Tables, and Footnotes," and Chapter 12, "Object Format Rules."

Graphics

There is no standard mechanism for representing graphics in markup data. There are several common methods in use, in each of which an entity holds the graphic object itself.

In markup, the entity can be in an external file, written in a particular format such as Sun raster format. The graphic data format is given a name called a *data content notation*. The entity declaration specifies its notation.

In XML, the graphic and its file format can be represented in a an unparsed entity. Then the XML document can use this entity as an attribute in a graphic element to include the graphic in the document.

FrameMaker provides tools for creating a graphic. Alternatively, your users can import an external graphic, either by copying it directly into your FrameMaker document or by referring to an external graphic. In the latter case, the graphic remains in an external file and the name of the file is associated with the document. FrameMaker recognizes a graphic in several file formats, such as MIF or Sun raster format. Because FrameMaker determines the format directly by reading the graphic, you don't need to tell it the format of a graphic explicitly. Hence, there is no need to attach names to the formats.

For information on translating graphics, see Chapter 19, "Translating Graphics and Equations."

Equations

As with graphics, markup has no standard mechanism for representing equations, while FrameMaker has a complete tool for working with them. Once created, however, equations in FrameMaker have characteristics very similar to graphics. For this reason, FrameMaker treats equations in essentially the same way as graphics for markup import and export, and this manual discusses them together.

For information on creating graphics and equations in FrameMaker, see the *FrameMaker User Guide*.

For information on translating equations, see Chapter 19, "Translating Graphics and Equations."

Tables

FrameMaker has a complex facility for creating and manipulating tables within a FrameMaker document, including several special element types for use in your EDD. Tables have parts such as a title, body, rows, and cells.

Markup data doesn't have a standard mechanism for representing tables. As a result, their representation is unique to each DTD. In practice, many DTDs use the *CALS* table model, which is similar to the table description supported by FrameMaker. Other DTDs can have element structure that is not automatically recognizable as a table, but that needs to format as a table.

When you create an EDD from a DTD, FrameMaker automatically recognizes the CALS table model and creates elements of the appropriate type. If you have a different table model in your DTD, you'll need to supply rules to identify the table structure.

For information on working with tables in FrameMaker, see the *FrameMaker User Guide*. For information on defining tables and table parts in an EDD, see Chapter 9, "Structure Rules for Containers, Tables, and Footnotes." For information on translating tables, see Chapter 18, "Translating Tables."

Cross-references

A *cross-reference* is a passage in one place in a document that refers to another place (a *source*) in the same document or a different document. While the XML and SGML standards do not explicitly support cross-references, they do provide the declared values `ID`, `IDREF`, and `IDREFS` for attributes, and attributes using these declared values customarily represent cross-references. FrameMaker can use this model for cross-references within a FrameMaker document.

FrameMaker provides several additions to the cross-reference model suggested by XML and SGML. You need to keep these possibilities in mind when you work with cross-references:

- For markup data, both the cross-reference and its source are elements. In FrameMaker the source of the cross-reference can be an element but can also be a paragraph or a spot in a paragraph.
- For markup data, attributes contain the information connecting a cross-reference to its source. In FrameMaker you can also store the information in markers instead.
- The `ID/IDREF` mechanism is natural in markup data for *internal cross-references*, those in which the source and the cross-reference are in the same document. However, it cannot be used with *external cross-references*, those in which the source and the cross-reference are in different documents. FrameMaker provides a single mechanism for both internal and external cross-references.

- FrameMaker represents external cross-references in XML with a variation of the `ID/IDREF` model. Instead of an `IDREF` attribute, it uses a `srcfile` attribute whose value identifies the file containing the source element as well as the value of its `ID` attribute.
- FrameMaker and markup have different views of what constitutes an internal or external cross-reference. In FrameMaker, a cross-reference to a different file is always an external cross-reference. In markup, cross-references to different entities in a single document are always internal cross-references. So cross-references between components in a FrameMaker book are considered external, but cross-references between the text entities that correspond to those components are internal, since the entire book translates to a single XML or SGML document.
- XML and SGML allow a single attribute value to link several sources. FrameMaker requires a separate cross-reference for each citation.

For information on creating cross-references in FrameMaker documents, see the *FrameMaker User Guide*. For information on translating cross-references to markup, see Chapter 20, "Translating Cross-References."

Processing instructions

Markup data can include *processing instructions* to modify the treatment of a document in some system-specific manner. FrameMaker translates most processing instructions as markers in your FrameMaker document. It also defines a small number of special processing instructions for handling FrameMaker markers, books, and book components. You can use the FrameMaker Developer's Kit (FDK) to handle other processing instructions when reading markup documents, but not when creating an EDD from a DTD.

For information on handling processing instructions, see Chapter 17, "Translating Entities and Processing Instructions," and Chapter 24, "Processing Multiple Files as Books."

Parameter entity declarations

A DTD can use parameter entities as references to declaration content that is repeated in different statements. When FrameMaker reads a DTD with parameter entities, it expands them out to the full content of the reference. Any declarations that use a parameter entity will translate with the correct content, but the resulting EDD does not retain the parameter entity declaration nor the parameter entity itself.

When you create an EDD, you can use FrameMaker variables as references to repeated content. When FrameMakertranslates the EDD to a DTD, it does not convert these variables to corresponding parameter entities.

4 The SGML and FrameMaker Models

Before building a structure application that is designed to work with SGML, you should understand the specific differences between SGML and the FrameMaker document model. For a discussion about markup in general, see Chapter 3, "A General Comparison of Markup and FrameMaker Documents."

SGML declaration

According to the SGML standard, an SGML document contains an *SGML declaration*, a *prolog*, and a *document instance set*. The SGML declaration includes information on the specific syntax in effect for the document. In the absence of an SGML declaration, your SGML applications can use the *reference concrete syntax* defined in the SGML standard.

SGML features with no counterparts

SGML features with no FrameMaker counterparts include:

- Short reference mapping or usage
- Markup minimization
- Content references

FrameMaker correctly interprets a document with markup using any of the features mentioned in this list. However, it does not retain the information that the document's markup used the feature. For example, your document might use *markup minimization* to omit certain start-tags. If it does, the parser interprets the document as though the omitted start-tags were present. The rest of FrameMaker's processing can't distinguish whether or not the start-tags were omitted.

FrameMaker also doesn't use any of the above features when writing SGML documents. If you want an SGML document written by FrameMaker to use one of the features, you must write a Structure API client. For information on writing a Structure API client, see the online manual that comes with the FDK, the *Structure Import/Export API Programmer's Guide*.

Marked sections and conditional text

Both SGML and FrameMaker have mechanisms for specifying portions of a document that can be included or left out as needed. In SGML, this mechanism is *marked sections;* in FrameMaker, it is *conditional text*. The details of these two mechanisms differ significantly. For this reason, when translating between FrameMaker and SGML, the software does not

attempt to preserve the fact that information was tagged as conditional text in a FrameMaker document or that it occurred in a marked section in an SGML document.

When reading an SGML document, if the SGML parser used by FrameMaker encounters a marked section declaration with the effective status `IGNORE`, it doesn't include that section. If the effective status is `INCLUDE`, `CDATA`, or `RCDATA`, the software appropriately interprets and translates the marked section. The software doesn't annotate marked sections in the resulting EDD or document. Since your modifications only affect documents after they have passed through the parser, you cannot modify this behavior.

Similarly, if FrameMaker encounters conditional text that is hidden when writing a FrameMaker document as SGML, it does not include that text in the SGML document. All other text, whether it is unconditional or conditional, is included in the SGML document. Conditional text is not annotated in any particular way in the resulting DTD or document. You can write a Structure API client to change the exported document instance to reflect condition tags.

For information on working with conditional text, see the *FrameMaker User Guide*.

Unsupported optional SGML features

The SGML standard defines some features as optional, meaning that a specific implementation does not have to accommodate these features to be considered a conforming SGML system.

The following optional SGML features are not supported by FrameMaker:

- `DATATAG`
- `RANK`
- `LINK`
- `SUBDOC`
- `CONCUR`

Your SGML documents cannot contain any of these features. If they do, FrameMaker signals an error and terminates processing. You cannot change this behavior with the FDK.

5 The XML and FrameMaker Models

Before building a structure application that is designed to work with XML, you should understand the specific differences between XML and the FrameMaker document model. For a discussion about markup in general, see Chapter 3, "A General Comparison of Markup and FrameMaker Documents."

Namespace declaration

FrameMaker supports the use of namespace declarations in XML. A namespace can be declared as an element attribute in the DTD, or it can be declared in a given element within the document instance.

FrameMaker does not require you to declare namespace attributes in the EDD.

In FrameMaker you can enable or disable namespaces in the XML structured application. To see namespace information, you can choose Element > Namespaces to display the NameSpaces dialog box. Or namespace information may appear in the structure view elements that declare namespaces. The display of namespace information in FrameMaker depends on whether namespaces are enabled, as well as how the namespaces are declared in the DTD.

For a structure application with namespaces enabled:

- When a namespace is declared in an element but not in the DTD, the namespace information appears in the Namespaces dialog box. The structure view shows an asterisk for the element that has the namespace declaration.
- When a namespace is declared in an element and is also declared in the DTD, the namespace information appears in the Namespaces dialog box. The structure view shows the namespace as a valid element attribute with no value. The structure view also shows an asterisk for the element.
- When a namespace is declared in the DTD, but is not declared in any element, the structure view shows the namespace as a valid attribute with no value.

For a structure application with namespaces disabled:

- When a namespace is declared in an element but not in the DTD, the structure view shows the namespace as an invalid attribute, with the URI as the attribute value.
- When a namespace is declared in an element and is also declared in the DTD, the structure view shows the namespace as a valid attribute, with the URI as the attribute value.

- When a namespace is declared in the DTD, but is not declared in any element, the structure view shows the namespace as a valid attribute with no value.

When you export the FrameMaker document to XML, the namespace information is written to the XML.

Rubi text

Japanese text uses a form of annotation known as Rubi to express the phonetic representation of a word over the characters used to spell the word. The XML specification suggests a representation for Rubi as follows:

```
<ruby>
   <rb>Oyamoji text</rb>
   <rt>Rubi text</rt>
</ruby>
```

To import that representation you would need read/write rules similar to the following:

```
element "ruby" is fm rubi group element "MyRubiGroup";
element "rb" is fm rubi element "MyRubi";
element "rt" is fm rubi element "Oyamoji";
```

For more information, see "Defining a Rubi group element" on page 97.

Element and attribute definition

The XML specification includes the following features concerning the declaration of elements and attributes:

Supported characters in element and attribute names

In XML, various markup tokens (GIs, attribute names, etc.) can include characters from the upper range of UNICODE. While FrameMaker can read XML content that contains the full range of UNICODE, FrameMaker has limited support of the characters for markup tokens. This support is limited to characters of the current display encoding. This display encoding can be determined by the locale of the current operating system, or by specifying a display encoding for the XML structure application. The characters you can use are:

Character set:	For this language:
FrameRoman	Western European languages
JIS X 0208:1997	Japanese
BIG5	Traditional Chinese
GB2312-80	Simplified Chinese
KSC5601-1992	Korean

For a given encoding, character support in markup tokens is further limited by the set of reserved characters. To see which characters are reserved for FrameRoman encoding,

refer to the online manual, *FrameMaker Character Sets* and note the reserved characters in the standard character set.

For more information, see "Specifying the character encoding for XML files" on page 68.

Multiple attribute lists for an element

The XML specification allows you to define an element's attributes in more than one attribute list. FrameMaker does not support this feature.

Unicode and character encodings

XML supports any character encoding that is compatible with UNICODE. FrameMaker offers equal support for any #PCDATA, CDATA and RCDATA content.

FrameMaker has limited support of characters in markup tokens such as GIs and attribute names—see "Supported characters in element and attribute names," (the previous section) and "Specifying the character encoding for XML files" on page 68.

Read/write rules do not support Asian characters, so you cannot use rules to manipulate elements or attributes whose names contain such text.

Supported encodings

FrameMaker ships with support for the following encodings, listed by their IANA names:

Big5	KSC_5601
EUC-CN	macintosh
EUC-JP	Shift_JIS
EUC-KR	US-ASCII
EUC-TW	UTF-16
GB2312	UTF-8
ISO-8859-1	windows-1252

Using these IANA names, you can specify any of these encodings for export to XML—see "Specifying the character encoding for XML files" on page 68.

These encodings are created in the ICU (International Components for Unicode) format, and stored as .cnv files. The supplied encodings are stored in the following locations:

- Windows: `$installdir\fminit\icu_data`
- Unix: `$installdir/fminit/icu_data`
- Macintosh: `$installdir:Modules:Icu_Data`

You can add other ICU encodings to the FrameMaker installation—you must create ICU compliant mappings and save them as .cnv files, then store them in the `icu_data`

directory. Once you install a new encoding, you can refer to it the same as you refer to the other export encodings.

FrameMaker associates a given ICU encoding to an internal display encoding that is appropriate for a given language. Because any number of languages can be used in an XML file, FrameMaker cannot make that association automatically. The XML document prolog specifies an encoding, but the document may contain elements or other constructs that override the language implied by that encoding.

As of this writing you can find more information about International Components for Unicode (ICU) at http://www-124.ibm.com/icu/.

FrameMaker display encodings

By default, FrameMaker uses the display encoding that matches the input locale of your operating system. However, you can specify a different encoding in your XML structure application. FrameMaker supports the following display encodings:

Display encoding:	For this language:
`FrameRoman`	Western European languages
`JISX0208.ShiftJIS`	Japanese
`BIG5`	Traditional Chinese
`GB2312-80.EUC`	Simplified Chinese
`KSC5601-1992`	Korean

On import, when FrameMaker encounters an unknown character in #PCDATA it imports the character as a marker of type `UNKNOWNCHAR`. The marker content is a text string representing the UTF16 encoding for that character. It imports unknown characters in CDATA as XML character references.

Encoding for XML files

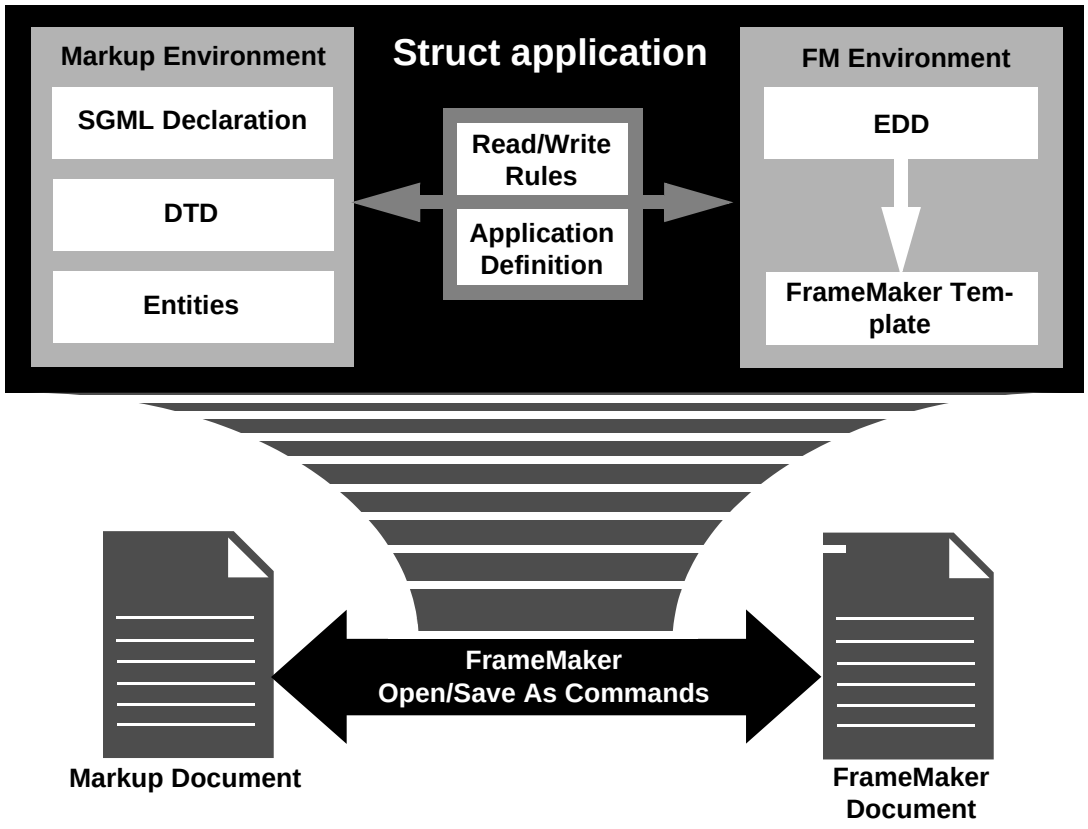
The XML specification states that an XML document must either specify an encoding in the prolog, or it must be UTF-8 or UTF-16. FrameMaker follows this specification by assuming UTF-8 by default if there is no encoding specified in the XML file.

If you read an XML file with character encoding that doesn't match either the declared encoding or the default encoding (if no encoding is declared), it is likely that the import process will encounter a character that doesn't match the encoding FrameMaker uses. In that case, you will get a parsing error that says the document is not well-formed due to a bad token.

6 Creating a Structure Application

This chapter describes the major tasks you'll perform to create a structure application and describes the pieces your application might contain. Much of your work in creating an application involves writing read/write rules to modify the software's default behavior. Part III, "Translating between markup data and FrameMaker," describes specific default behaviors and explains how you can change them.

The diagram shows your completed structure application at work. Information in the application funnels into the Open and Save As commands in FrameMaker, adapting their default translation behavior to your markup and FrameMaker environments. Your structure application has information (such as a DTD) specific to the markup representation of documents, other information (such as an EDD) specific to the FrameMaker representation of documents, and rules and definition to bridge this information. The application pieces are described in more detail in "Pieces of a structure application" on page 41.



The development process

As a structure application developer, your primary task is to modify the default translation behavior of FrameMaker to fit the circumstances of your application. This section gives an overview of the steps used to create a structure application and to deliver that application to your end users.

At the highest level, there are four major application development tasks:

1. Get an initial version of both an EDD and a DTD.

   You typically start application development with either an existing EDD or an existing DTD. In some situations, however, you have neither of these things. Your first task is to provide yourself with either an EDD or a DTD as a starting point and then use the software to create an initial version of a DTD if you started with an EDD or an EDD if you started with a DTD.

2. Get sample documents to test your application.

   If you don't have sample markup documents and FrameMaker documents to test the progress of your application, you will need to create them.

3. Create read/write rules to modify the translation between the EDD and DTD.

   Once you have an EDD, a DTD, and some sample documents, start an iterative process of analysis and application modification. Write read/write rules to modify how the software translates between markup and FrameMaker.

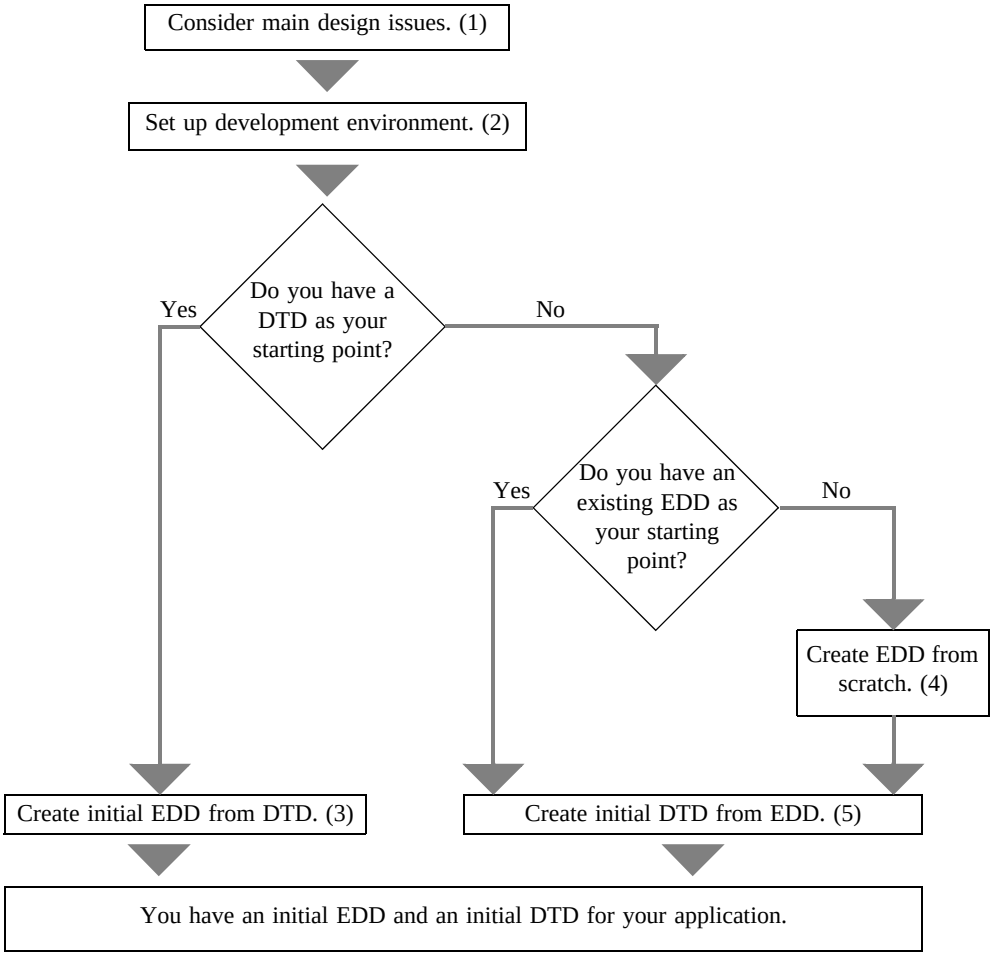
4. Use the FDK and FrameMaker tools to finish your application.

   Once you've done as much as you can with rules, you may have to write a *structure API client* to further modify the software's behavior. Consider other modifications you want, such as customizing the FrameMaker end-user interface to remove inappropriate commands or add new ones. Finally, assemble all the pieces of your application for delivery to your end users.

The four following sections provide more detail on the major tasks. To simplify the discussion, they have no pointers to more information on the various steps. For these pointers, see "For more information" on page 40.

Task 1. Producing an initial EDD and DTD

This flowchart shows the steps you follow to produce an initial EDD and DTD for your application, and the notes below the chart give more detail on some of the steps. What you do once you have the development environment set up depends greatly on the situation at your company. Numbers in the flowchart refer to the notes below the chart.



The following notes give more detail on some of the steps in the chart. Note numbers match the chart numbers.

(1) Consider the main design issues.
You need to think about:

- the kinds of documents you need to support
- what end users need in terms of access to both FrameMaker and markup versions of the same document
- the environment (XML, SGML, or FrameMaker) in which documents will be created and delivered
- whether end users work with only one structure application or multiple applications

(2) Set up the development environment.
You or your system administrator must install FrameMaker.

Once FrameMaker is properly installed, you need to tell it where to find rules you'll write and other information it may need. Collect this information in an application definition that you can associate with the EDD or the markup document element.

At this point, you can create a simple application file, giving the application a name and specifying the location of existing files, such as an EDD, DTD, and (for SGML structure applications) the SGML declaration. Later in the development process, you'll need to include other information in the definition.

(3) Create an initial EDD from your DTD, if you have one.
If you're starting with a DTD, choose File>Developer Tools>Open DTD to create an initial EDD to see how the software translates your DTD with no help from your application. You use this initial EDD during your analysis to see how you want to translate markup constructs into FrameMaker constructs.

In the absence of read/write rules, FrameMaker translates:

- Markup elements to FrameMaker elements of the same name—For SGML the FrameMaker element names are receive initial capitalization; for XML the FrameMaker elements receive the case found in XML
- Markup attributes to FrameMaker attributes, assuming that attributes contain extra information about their associated elements
- entities to various constructs such as variables

The software produces a log file if it encounters any problems while processing your DTD.

(4) Create an EDD, if you're starting from scratch.
If you're in the situation of having neither a preexisting EDD nor a preexisting DTD, you need to create one or the other before you can create a structure application. Because of the richer semantics available in an EDD, you should first create an EDD and its associated FrameMaker template and then continue the development process from there.

If you're starting from scratch and the sample documents you intend to use are unstructured FrameMaker documents, you may want to do this step in conjunction with the next major task.

(5) Create an initial DTD from your EDD.
If you have a preexisting EDD, choose File>Developer Tools>Save As DTD to create an initial DTD to see how the software translates your EDD with no help from your application.

The DTD has element and attribute definition list declarations corresponding to the element and attribute definitions in the EDD. The software reads through the EDD, processing the elements and their attributes one at a time. For each definition, the software

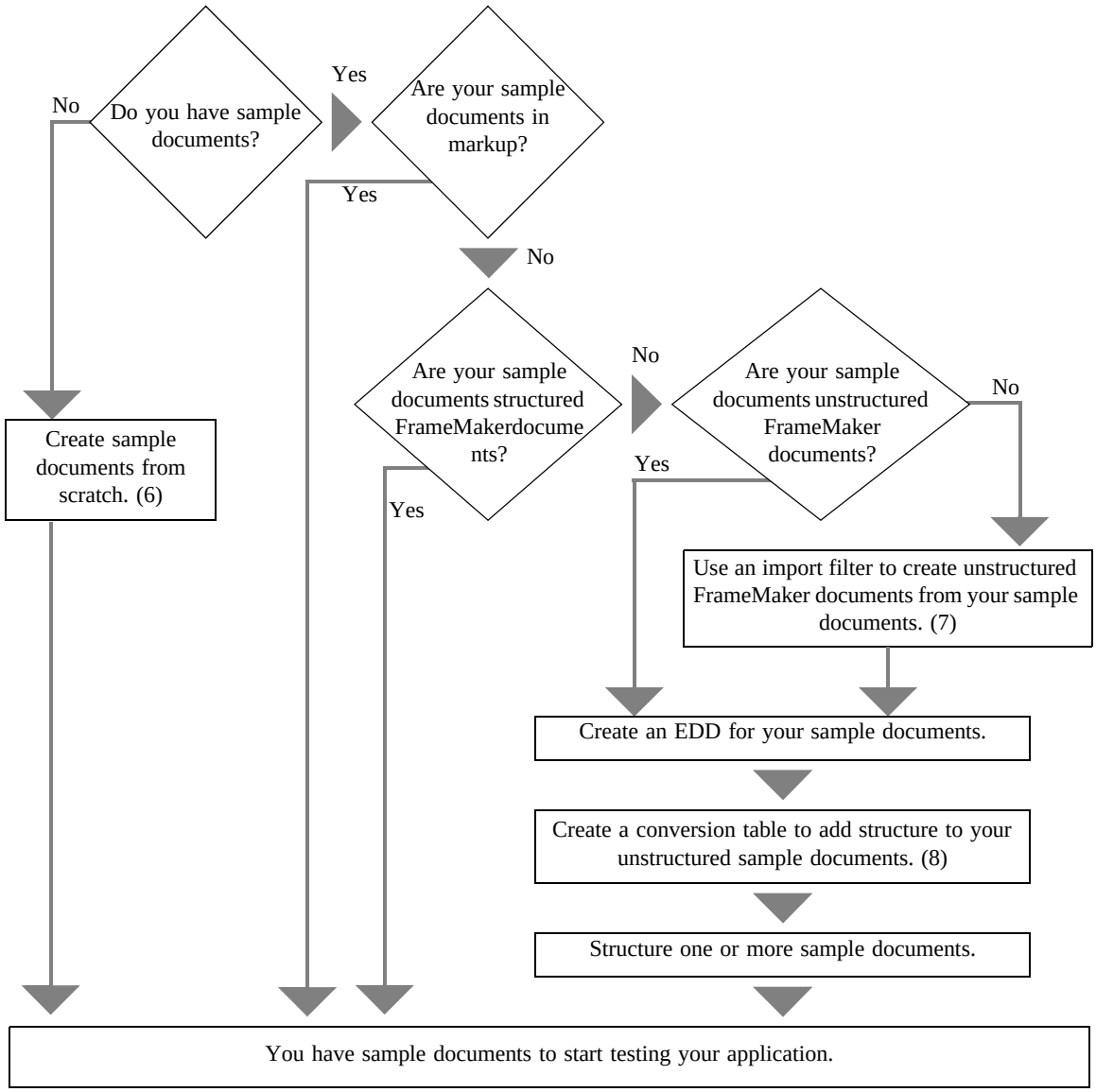
- creates a markup element of the appropriate type
- produces an attribute definition list declaration for any element that has defined attributes

- writes notation declarations for use with graphics and equations and produces comments in the DTD corresponding to those in the EDD

An EDD includes more semantic information on the usage of its elements than does a DTD. For example, there are special element types corresponding to markers, system variables, and graphics, among other things. The declarations in the DTD created by FrameMaker reflect this information.

Task 2. Getting sample documents

This flowchart shows the steps you follow to get sample documents to test with your application, and the notes below the chart give more detail on some of the steps. Numbers in the flowchart refer to the notes below the chart.



The sample documents you need depend on your starting point. For example, if you already have markup documents, you probably won't yet have FrameMaker documents. If you have existing unstructured FrameMaker documents, you may need to structure them. Later in the process, you may decide to create more sample documents.

(6) Create sample documents if you have none.
Add to the collection as you develop the application, in order to test particular parts of the application. If you're starting with a preexisting DTD, create sample XML or SGML

documents. If you're starting with a preexisting EDD, create structured FrameMaker documents.

(7) Use document import filters to get FrameMaker documents.
If your sample documents are in a file format other than XML, SGML, or FrameMaker, you should convert them to unstructured FrameMaker documents and then use the software's tools for structuring unstructured documents.

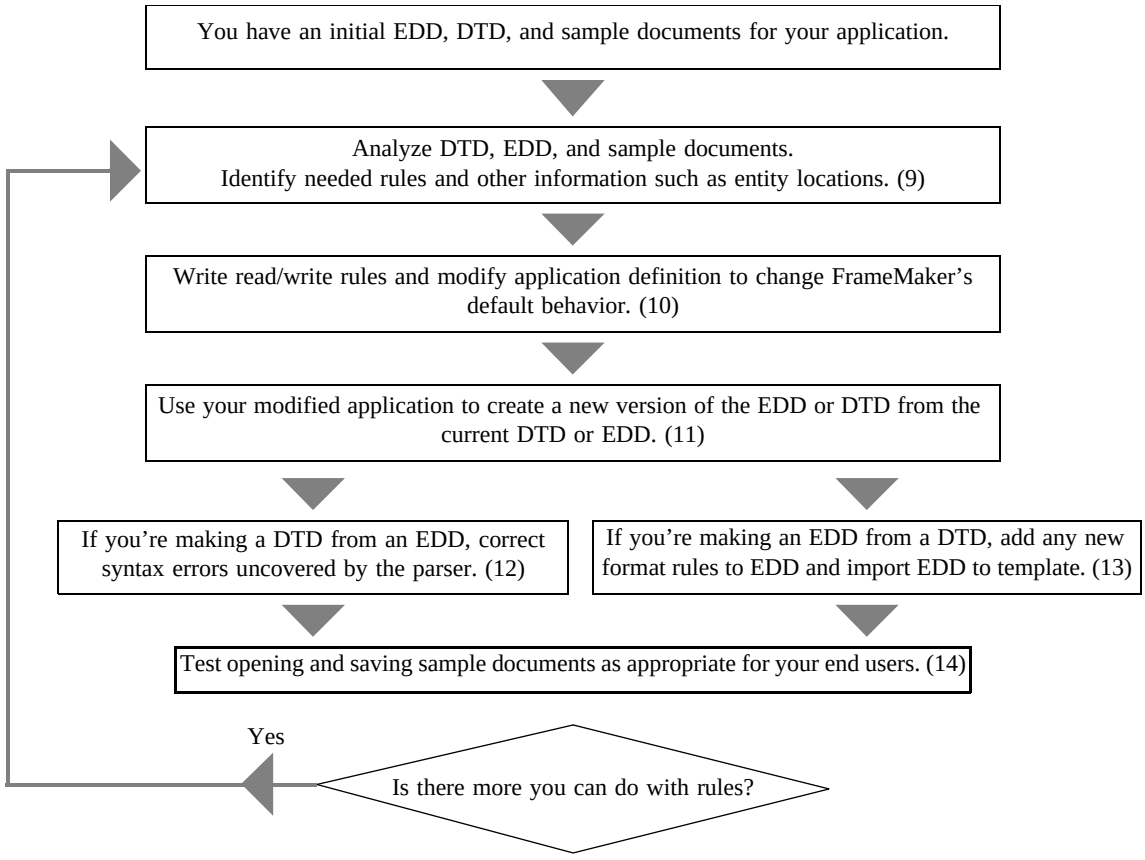
(8) Structure unstructured FrameMaker documents if necessary.
If you've gotten to this point, you have unstructured FrameMaker documents to use as your sample documents. FrameMaker provides tools to aid you in structuring unstructured documents.

You create a conversion table that describes a set of unstructured documents and then use FrameMaker commands to structure your documents. Once the documents are structured, you also need an EDD and structured template that corresponds to the elements described in the conversion table.

If you use FrameMaker's conversion tools, you should use the resulting EDD as one of the starting points of your structure application.

Task 3. Creating read/write rules

Once you have both an EDD and a DTD, you can use read/write rules to refine how FrameMaker translates between them. This portion of application development is an iterative process. This flowchart shows the steps of this process. The notes below the chart give more detail on some of the steps, and numbers in the flowchart refer to the notes.



(9) Analyze the details of what your application must do.
If you are not already familiar with it, you need to study the original DTD or EDD, with the partially completed new EDD or DTD and sample documents, to determine what constructs are defined and how they are intended to be used. Sample documents are invaluable in understanding the meaning of the defined element structures; the partially completed EDD or DTD lets you know how well FrameMaker performs the translation so far.

If you start with a DTD, the steps of your analysis should be similar to the following:

1. Examine element declarations or definitions.

   Determine the representation of elements and, from a high level, the element and document groupings. Determine which elements are used for text or as containers for other elements and which are used for special constructs such as tables or graphics.

   Determine the level of minimization provided by the DTD. *Markup minimization* is irrelevant in FrameMaker's internal representation, but you may need to understand its usage in sample documents. During the early stages of development, you probably won't try to reproduce minimization in markup, but you may later write a structure API client to do so.

Determine the purpose of elements with a declared content of EMPTY. These will translate to the appropriate FrameMaker constructs, such as graphic elements.

2. Examine attribute use.

   Examine attributes to distinguish attributes that control formatting from attributes that simply provide information about the associated element. Formatting attributes for special constructs such as tables or graphics may become formatting properties in a FrameMaker document.

3. Examine entity declarations.

   Determine how entities are used and which become text, variables, graphics, book files, or special characters in FrameMaker. The SGML standard defines several sets of character entities. Check whether your DTD refers to any of these character sets.

4. Examine notation declarations.

   Determine how to represent non-SGML data (NDATA) in FrameMaker. NDATA can often be represented directly as FrameMaker graphic or equation elements.

5. Examine elements and attributes used as cross-references.

   FrameMaker assumes markup attributes with declared value of ID or IDREF refer to cross-references. Your DTD may also use other declared values, such as IDREFS, NAME, and NAMES, for cross-references.

(10) Write rules and modify the application definition.
While working on your rules, you may need to modify your application definition. For example, you need to tell the software how to locate entities that have *public identifiers*.

Some parts of the task of changing the translation between the two representations will require more work than others. For example, standard elements such as those representing text in paragraphs or lists may translate with no help from you. But other elements, such as those representing graphics, will require your assistance.

The steps described here suggest working on a portion of the representation only as far as possible with rules and waiting until you're finished with rules before attempting to write a structure API client. An alternate approach is to work on a particular representation until it is complete, before moving on to the next representation, even if that means creating a structure API client.

(11) Create a new version of the EDD or the DTD.
After you have written the rules your application requires, use the appropriate command to update your EDD or to recreate your DTD. Where they are applicable, FrameMaker uses your rules to modify its processing. Where it encounters no applicable rule, the software uses its default behavior.

(12) Correct syntax errors uncovered by the parser.
FrameMaker uses parsers to interpret XML and SGML markup in ways that comply with the associated standards. When FrameMaker processes a DTD or a markup document on

import, the parser interprets markup and identifies the information in the document such as element declarations and element start-tags.

The parser may find problems that cause the DTD to be syntactically invalid. Some problems can be corrected with read/write rules, while others may require you to modify the DTD.

SGML: The SGML standard allows wide variations in the actual markup of specific documents. For example, it allows variations in the maximum length of the sequence of characters specifying a name, in the case-sensitivity of names, and in markup that may be omitted altogether. This is specified in the SGML declaration. If the parser encounters errors in the DTD, you may be able to fix them by changing the quantity and capacity limits defined in the SGML declaration used by the current structure application.

Modify the SGML declaration using a text editor to correct these errors. Add the SGML declaration to the application definition. Rather than recreating the DTD with the modified application, you can manually invoke the SGML parser to validate your changes.

(13) Add format rules to the EDD and import it into the template.
If you started with a DTD, you (perhaps working with your document designer) now expand the EDD and FrameMaker template to include appropriate format rules.

FrameMaker supports context-sensitive formatting. An element definition can have one or more format rules that specify formatting applied to the element in a particular context in the document. For example, a `Head` element inside a single `Section` element might be formatted differently than a `Head` element inside a `Section` element that's nested in another `Section` element. (That is, a first-level heading probably looks different from a second-level heading.)

(14) Test with sample documents.
You should test your application at various stages of its development and you should test your read/write rules thoroughly.

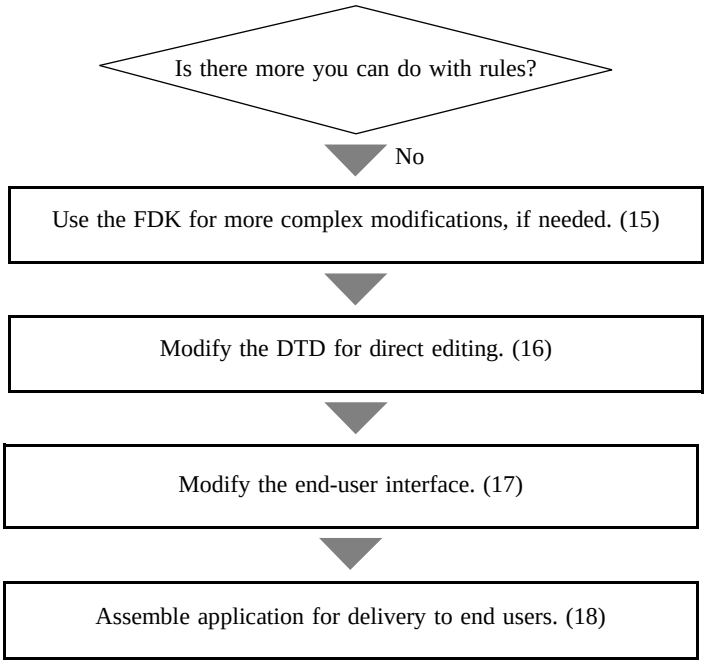
If end users will be opening markup documents in FrameMaker, invoke the software with sample documents that test your modifications in that direction. If users will be saving FrameMaker documents as markup, invoke the software with sample documents to test those modifications. If your end users have existing documents for you to use, that can be very helpful. In any case, you may want to create sample documents that test most features of your application.

Task 4. Finishing your application

Once you've progressed as far as you can with read/write rules and the application definition, you may find that FrameMaker still does not translate your documents as completely as you wish. If so, you'll have to use the FDK to complete the translation. At this time, you may also decide to modify FrameMaker's end-user interface to be more

appropriate to your end users. Finally, you put all the pieces of the application together to deliver to your end users.

The flowchart shows these steps. The notes below the chart give more detail on some of the steps, and numbers in the flowchart refer to the notes.



(15) Use the FDK for more complex modifications.
In some situations, read/write rules are not sufficient to describe the appropriate translations between markup and FrameMaker. For example, your SGML DTD may use special markup minimization, and rules cannot specify creation of minimized markup during export of documents to SGML. In this case, you'd need to write a structure API client.

Even if you need to write a structure API client, you should use read/write rules for as much of the translation as possible. This will simplify what your client must do.

(16) Modify the DTD for direct editing.
If you've created a DTD from an EDD and your end users will be editing markup documents directly, you may want to make modifications to the DTD for ease of use. For example, you may choose to change some element definitions to allow tag omission. Furthermore, you may wish to add short reference mapping or to use additional declarations in the DTD to make it easier for your end users.

(17) Modify the end-user interface.
You can modify the menu items available to your end users. At the very least, you'll probably want to remove the Developer Tools submenu from the File menu. This submenu contains only developer commands. You may wish to remove other menu items as well. For

example, if you want all formatting controlled by the format rules in your EDD, you can remove direct formatting commands from your users' menus. In addition, you might write FDK clients to help your end users in other ways and provide access to those clients on the menus.

(18) Assemble your application for delivery to end users.
You need to make sure the pieces of your application are appropriately packaged for your end users. You may even choose to write scripts to make it easier for your end users to access the application. You should also be prepared to update your application as end users encounter bugs or request enhancements. In addition, be prepared to handle revisions of the EDD or DTD.

For more information

You can find more information on topics in the previous section in other parts of this manual and in other manuals. The table lists the topics by the notes that introduced them.

Note	Topic	Location
2	Installing FrameMaker	*Installing FrameMaker*
	Creating an application definition	"Application definition file" on page 50
3, 4	Creating an EDD	Part II, "Working with an EDD"
	Working with log files	"Log files" on page 71
7	Using document filters available with FrameMaker	*Using Filters*
8	Creating an EDD and FrameMaker template	Part II, "Working with an EDD"
	Creating an EDD for an existing set of unstructured documents	Appendix A, "Conversion Tables for Adding Structure to Documents"
9-14	How FrameMaker translates markup documents and how you can modify that with rules	Part III, "Translating between markup data and FrameMaker"
	Writing format rules	Chapter 10, "Text Format Rules for Containers, Tables, and Footnotes," and Chapter 12, "Object Format Rules"
	Importing an EDD into a FrameMaker template	Chapter 8, "Developing an Element Definition Document (EDD)"
15	Using the FDK to create a structure application	*Structure Import/Export API Programmer's Guide*
17	Modifying the user interface	*Customizing FrameMaker Products* (for Macintosh or WIndows®) or *Changing Setup Files* (for UNIX®)
18	Assembling your application	"Pieces of a structure application," next

Pieces of a structure application

A structure application created with FrameMaker includes an external DTD subset, a FrameMaker template, and a read/write rules document. Other files may be included, depending on the particular application.

The following sections describe the possible parts of a structure application and your options for assembling those parts into a working application. Although you can include a structure API client as part of your application, this section does not talk about what to do with such a client.

Application definition file

Your application requires several files, search paths for these files, and other information. You provide FrameMaker with these paths and other information by placing them in the `structapps.fm` file. In this file, you create a definition for your application that includes the names of other needed files.

If you provide your users with several structure applications, put information about all of them in a single `structapps.fm` file. FrameMaker reads this file on startup, so your end users can choose any applications defined in this file. You need to deliver this `structapps.fm` file to your end users.

For information on the `structapps.fm` file, see "Application definition file" on page 50.

External DTD subset

A structure application pertains to a particular external DTD subset that the software uses when writing markup documents. You specify the external DTD subset in the application definition. While it is optional, to take full advantage of FrameMaker we recommend that you specify a DTD for your application. Your application will use it to:

- Export a FrameMaker document as markup.
- Import a partial markup instance as a text inset in your FrameMaker document.
- Import an entity reference to a markup file as a text inset in your FrameMaker document.
- Open an incomplete markup instance as a FrameMaker document.

SGML declaration

> **XML:** The XML specification does not support user-defined SGML declarations. The following discussion is for SGML, only.

An SGML structure application can use a particular SGML declaration. If it does, you specify the SGML declaration in the application definition.

The SGML declaration is an optional part of an application. Individual SGML documents can begin with a declaration. If neither the SGML structure application nor a particular SGML

document specifies a declaration, FrameMaker uses the SGML declaration described in Appendix D, "SGML Declaration."

If you do specify an SGML declaration, you can deliver it to end users as a separate file.

FrameMaker template

A structure application is associated with a particular FrameMaker template that you can specify in the application definition. The software uses the template to specify the structure and formatting of FrameMaker documents it creates from markup documents. If you do not specify a template, it uses the formats you would get if you used the New command and specified Portrait, rather than a particular template.

As you create your application, you can work with a FrameMaker EDD in a separate file. However, FrameMaker does not use a separate EDD file to find information on structure when opening a markup document; it gets the information directly from the FrameMaker template. At some point during the creation of the FrameMaker template, you must import element definitions from the EDD to the template. Because of this, you don't deliver the EDD to your end users as a separate file. For maintenance purposes, however, you should retain the EDD in a separate file.

Read/write rules document

You create read/write rules in a FrameMaker document and specify that document in the application definition. FrameMaker uses the rules document both when opening a markup document and when saving a FrameMaker as XML or SGML.

Because a read/write rules document can reference other documents, you can have more than one document containing rules. In the application definition, you specify only the primary file; the software locates the others through the `#include` mechanism in the primary file.

Your read/write rules document and any include files are separate files you deliver to your end users.

Entity catalogs

If your application requires special external entities or entity catalogs, you must deliver those as separate files to your end users. For information on entities and entity catalogs, see Chapter 17, "Translating Entities and Processing Instructions," Chapter 7, "Working with Special Files," and Appendix F, "ISO Public Entities."

Documentation

As part of your structure application, you should write documentation to explain to your end users how to use the application. It is your responsibility to deliver such documentation either in online or printed form.

This manual is written for developers creating structure applications. Refer end users to the FrameMaker and FrameMaker user guides. You can remove this manual from your end users' installation directory.

Creating a FrameMaker template

You should deliver your FrameMaker template to end users as a separate file. The template is a FrameMaker document that includes element definitions and formatting information, but not the ultimate content of your documents. By using a template, you ensure all documents for a specific application can have the same element definitions and formatting. For information about importing element definitions into your template, see "Creating an Element Catalog in a template" on page 103

For information on creating a FrameMaker template, see the FrameMaker and FrameMaker user guides. For information on creating the EDD for a template, see Chapter 8, "Developing an Element Definition Document (EDD)."

Following are descriptions of information you can put in your template that takes advantage of document structure. Included are descriptions of building blocks that define cross-reference formats, variable definitions, and format generated lists and indexes, as well as map FrameMaker elements for HTML export. For general information about cross-reference formats, variable definitions, generated lists and indexes, and HTML export, see the *FrameMaker User Guide*.

Cross-reference formats

Use the following building blocks to create cross-reference formats that refer to FrameMaker elements:

Building block	Meaning
<$elempagenum>	The page number of the source element
<$elemtext>	The text of the source element (up to the first paragraph break), excluding its autonumber, but including any prefix and suffix specified in the element definition
<$elemtextonly>	The text of the source element (up to the first paragraph break), excluding its autonumber and any prefix and suffix specified in the element definition
<$elemtag>	The tag of the source element
<$elemparanum>	The entire autonumber of the source element's first paragraph (or of the paragraph containing the source element), including any text in the autonumber format

Building block	Meaning
<$elemparanumonly>	The autonumber counters of the source element's first paragraph (or of the paragraph containing the source element), including any characters between the counters
<$attribute[name]>	The value of the attribute with the specified name (or, if no value is specified, the default value)

You can specify building blocks to refer to the *ancestor* of the source element. Including the tag of an element in the building block indicates that it will refer to the nearest ancestor with the specified tag. For example, a cross-reference to a subsection might also identify its parent section, as in the following: See “Types of plate boundaries” in “Plate tectonics.”

The following list shows how building blocks refer to ancestors of the source element:

Building block	Meaning
<$elempagenum[tag]>	The page number of the nearest ancestor with the specified tag
<$elemtext[tag]>	The text of the nearest ancestor (up to the first paragraph break) with the specified tag, excluding its autonumber, but including any prefix and suffix specified in the element definition
<$elemtextonly[tag]>	The text of the nearest ancestor (up to the first paragraph break) with the specified tag, excluding its autonumber and any prefix and suffix specified in the element definition
<$elemtag[tag]>	The tag of the nearest ancestor with the specified tag
<$elemparanum[tag]>	The entire autonumber of the first paragraph of the nearest ancestor with the specified tag
<$elemparanumonly[tag]>	The autonumber counters of the first paragraph of the nearest ancestor with the specified tag, including any characters between the counters
<$attribute[attrname:tag]>	For the preceding elements that are first siblings, then ancestors of the source element, the value of the attribute with the specified name (or, if no value is specified, the default value)

In each of the building blocks, enter the tag of the element to which you want to refer between brackets. For example, if you want to refer to the text of the source's nearest ancestor tagged Section, you would use:

```
<$elemtext[Section]>
```

Variables

If you're defining running header/footer variables that will refer to elements in structured FrameMaker documents, you can use building blocks that refer to elements or element attributes rather than to paragraphs.

The following building blocks in running header/footer variables refer to an element tag:

Building block	What it displays
<$elemtext[tag]>	The text (up to the first paragraph break), excluding its autonumber, but including any prefix and suffix specified in the element definition
<$elemtextonly[tag]>	The text (up to the first paragraph break), excluding its autonumber and any prefix and suffix specified in the element definition
<$elemtag[tag]>	The tag
<$elemparanum[tag]>	The entire autonumber of the element's first paragraph
<$elemparanumonly[tag]>	The first paragraph's autonumber counters, including any characters between them

Follow these guidelines for using element tag building blocks:

- Enter the tag of the element for which you want to display information between brackets.
- You can include a context label with the element tag to provide additional information about the element's location in the document structure.
- You can include more than one element tag in the brackets, separated with commas. With multiple tags, FrameMaker uses the first element it finds with one of the tags. An example building block is:

  ```
  <$elemtext[Head(Level1),Head(Level2)]>
  ```

The following building blocks in running header/footer variables refer to element attribute values:

Building block	What it displays
<$attribute[name]>	The value of the attribute
<$highchoice[name]>	The highest value of the attribute that appears on the page (where highest means the value closest to the bottom of the pop-up menu on the right side of the Attributes window)
<$lowchoice[name]>	The lowest value of the attribute that appears on the page (where lowest means the value closest to the top of the pop-up menu on the right side of the Attributes window)

Follow these guidelines for using element attribute value building blocks:

- Enter the name of the attribute whose value you want to display between brackets. If a list of possible values is not defined for the attribute used in a <$highchoice> or <$lowchoice> building block, the building block will be ignored.
- You can specify elements to consider when searching for an attribute value to include in the running header or footer. To do so, place a colon after the attribute name, followed by one or more element tags separated by commas. For example, a variable with the following definition would display the highest value of the Security attribute of the first-level and second-level Section elements.

```
<$highchoice[Security:Section(Level1), Section(Level2)]>
```

Special text flows to format generated lists and indexes

Many formatting aspects of a list or index are controlled by a special text flow on a reference page in the generated file. The name of the reference page matches the default filename suffix, such as TOC for a table of contents or IX for a standard index. These reference pages contain building blocks to specify formatting of the entries in the generated files.

The following building blocks apply only to structured documents in FrameMaker. If you use an element-based building block to refer to an unstructured paragraph, the information won't appear in the generated list. If you use a paragraph-based building block to refer to an element, the information included in the generated list will come from the paragraph that contains the beginning of the element.

Building block	What it displays
<$elemtextonly>	Displays the text of the first paragraph of the element, excluding any autonumber, and the element's prefix and suffix, if any. Note that prefix and suffix are only excluded for the specific element; if the paragraph text is in a child element that has a prefix or suffix, the prefix or suffix will be included.
<$elemtext>	Displays the text of the first paragraph of the element, excluding any autonumber, but including the element's prefix and suffix, if any.
<$elemtag>	Displays the element tag

For general information about how FrameMaker generates and formats lists and indexes, see the FrameMaker user's manual.

HTML mapping for export

The *FrameMaker User Guide* includes instructions for converting a FrameMaker document to *HTML*. Converting FrameMaker documents to HTML uses nearly the same process, but there are some differences which take advantage of the structure in your documents.

When setting up an application, you should set up the mapping to HTML in advance so your users don't need to do that work. After you build a template, you can use the process described in the FrameMaker user's guide to set up the HTML mapping for it. The mapping information will be stored on the HTML reference page of your template.

You can also associate HTML mapping with an EDD. To do this, create a reference page in your EDD named EDD_HTML. Then copy the mapping information from the HTML reference page of your template to the EDD_HTML reference page. If your EDD includes the `ImportHTMLMapping` element, then the mapping information will be copied to any document that imports your EDD. For more information about including mapping information with an EDD, see "Specifying whether to transfer HTML mapping tables" on page 89

Mapping FrameMaker elements to HTML elements

You specify how to map FrameMaker elements and attributes to HTML elements in the HTML Mapping table on the HTML reference page. For example, in the following table, Section elements map to HTML `<Div>`, and Head elements map to HTML `<H1>` through `<H6>`:

FrameMaker Source Item	HTML Item		Include Auto#?	Comments
	Element	New Web Page?		
E:Section	Div	1,2,3	N	
E:Head	H*		Y	
E:List	Ol		N	
E:ListItem	Li		N	
A:Description	alt			

In the above example, the following syntax is used to specify FrameMaker elements and attributes in the column titled FrameMaker Source Item:

- `E:`*`elementname`*, where *`elementname`* is the name of the element
- `A:`*`attributename`*, where *`attributename`* is the name of the attribute

Note the following points when mapping a structured document to HTML:

- In unstructured documents, there is a Headings reference page which maps specific paragraph formats to HTML headings, and assigns a hierarchy to them. In a structured document there is no need for the Headings reference page. You map FrameMaker elements to HTML headings in the Mapping table. The hierarchy is determined by the context of the FrameMaker element, just as context can determine the formatting of a FrameMaker element.
- The document can be divided into separate web pages at certain levels of hierarchy. In the above example, HTML export will create a new web page for every section of a level 1, 2, or 3.

- In unstructured documents, you map paragraph formats to list items, and specify the list type for that list item's parent. In a structured document, you specify the list type for FrameMaker list element, and then you map FrameMaker list items to the HTML <Li> element. Also, you do not specify the nesting of lists, since that can be determined by the hierarchy in the FrameMaker document. In the above example, List maps to HTML <Ol>, and ListItem maps to HTML <Li>. If ListItem contains a List, on export to HTML the lists will nest correctly.
- In the above example, the Description attribute in FrameMaker maps to the alt attribute for HTML elements. Every occurrence of a Description attribute will result in an alt attribute in HTML, whether it is valid for that HTML element or not. The HTML specification says that User Agents should ignore attributes they don't understand, so this is not a major problem unless you need to validate your HTML file. For HTML elements that use the alt attribute (<IMG>, for example) your description can appear in place of that element.

Building blocks for structured documents

You can create HTML conversion macros that use information specific to elements and attributes. The following building blocks can be used when defining macros for elements and cross-reference formats:

Building block	Meaning
<$elemtext>	The text of the source element (up to the first paragraph break), excluding its autonumber, but including any prefix and suffix specified in the element definition
<$elemtextonly>	The text of the source element (up to the first paragraph break), excluding its autonumber and any prefix and suffix specified in the element definition
<$elemtag>	The tag of the source element
<$elemparanum>	The entire autonumber of the source element's first paragraph (or of the paragraph containing the source element), including any text in the autonumber format
<$elemparanumonly>	The autonumber counters of the source element's first paragraph (or of the paragraph containing the source element), including any characters between the counters
<$attribute[name]>	The value of the attribute with the specified name (or, if no value is specified, the default value)

7 Working with Special Files

This chapter discusses the special files you need to create a structure application. It tells you where to look for them in your directory tree; describes the workings of two of them, the application definition file and the log file; and points you to the appropriate chapters for information on the other files.

Location of structure files

FrameMaker provides a location for putting all of the special files associated with structure applications. Some files are provided with the software. Others are files you create and store at this location.

If you use FrameMaker on a UNIX platform, when the software needs one of these files it looks in several places. It searches directories in this order:

- *curdir*/fminit/*uilanguage*/structured where *curdir* is the directory from which you start FrameMaker and *uilanguage* indicates a particular user-interface language, such as usenglish or ukenglish
- fminit/*uilanguage*/structured in the user's home directory
- $FMHOME/fminit/*uilanguage*/structured, where $FMHOME is the directory in which FrameMaker is installed

If you use FrameMaker on a Macintosh or Windows platform, when the software needs one of these files it looks in a directory named structure in the home directory for FrameMaker. On a Windows platform, you can specify an alternate structure directory in the maker.ini file. For information about the maker.ini file, see the FrameMaker user's manual.

On all platforms, you can use the variable $STRUCTDIR to refer to the structure directory. The rest of this chapter follows this convention.

When you install FrameMaker, the structure directory contains the following files and subdirectories:

default.rw	The default read/write rules file. FrameMaker uses this file in the absence of a rules file in the current application. It also uses the file as the template when you create a new rules file with the Developer Tools>New Read/Write Rules File command on the File menu.
structapps.fm	The default version of the application definition file. You edit this file so that it contains the definitions of any applications you create.

`entfmts`	Default formats used for ISO public entities. For information on this file, see Appendix F, "ISO Public Entities."
`sgml/`	A directory containing SGML structure applications:
`sgml/docbook/`	The SGML DocBook starter kit files. For information on this directory, see the online manual *Using the DocBook Starter Kit*.
`sgml/isoents/`	Other ISO public entity files. For information on the contents of this directory, see Appendix F, "ISO Public Entities."
`xml/`	A directory containing XML structure applications.
`xml/xdocbook/`	The XML DocBook starter kit files. For information on this directory, see the online manual *Using the XDocBook Starter Kit*.
`xml/isoents/`	Other ISO public entity files. For information on the contents of this directory, see Appendix F, "ISO Public Entities."
`xml/xhtml/`	The XHTML starter kit files. For information on this directory, see the online manual Using the XHTML Starter Kit.

If you choose, you can create other directories under `$STRUCTDIR` to hold files for your applications.

Application definition file

You need to deliver a `structapps.fm` file to your end users. In some situations, you make more than one application available to your end users at the same time. For example, your end users may sometimes need an application based on the DocBook DTD for writing technical documentation and may at other times need an application based on a completely different DTD particular to your company. If so, you provide two applications in one `structapps.fm` file.

The `structapps.fm` file provides definitions for the available applications; it can also contain information relevant to all applications, such as a default place to look for entity files.

> **Important:** Reinstalling FrameMaker can overwrite the current `structapps.fm` file with the default version that ships with the product. Be sure your end users understand this. To keep your site-specific structure applications file intact, they should make a copy of this file before reinstalling, and then use the copy to overwrite the fresh version of `structapps.fm`.

FrameMaker can associate a particular document with a structure application in several ways:

- An EDD can explicitly associate itself with a structure application. If it does, all FrameMaker documents that use that EDD are automatically associated with the application.
- A structure application can name one or more document types (identified in the markup in the doctype declaration and in the document's root element). If it does, all markup

documents that use one of those document elements are automatically associated with the application.

- If neither of the above occurs or if the associations are ambiguous, FrameMaker asks the end user to pick a structure application to use with a document.

Editing an application definition file

The `structapps.fm` file is a structured FrameMaker document. You edit this file by choosing Structure Tools>Edit Application File from the File menu and using standard FrameMaker editing techniques. FrameMaker reads this file on startup. While developing your application, you may need to change its contents. If you do so, have the software reread the file by using the Structure Tools>Reread Application File command.

The Reread Application File command rereads the current `structapps.fm` file that is open and has current focus. This command replaces the current list of structure applications stored in memory with the applications defined in the current `structapps.fm` file. This is true for all platforms.

On UNIX, when you edit the `structapps.fm` file, you may choose to write it to a directory other than the one in which FrameMaker found it. When you then use the Reread Application File command, the software again looks in the three directories mentioned in "Location of structure files" on page 49.

Contents of structapps.fm

The highest-level element in an `structapps.fm` file is `StructuredSetup`. That element's first child must be `Version`, to indicate the product version. The `Version` element is followed by zero or more `SGMLApplication` or `XMLApplication` elements, each of which defines the pieces of a structure application. Finally, there can be an optional `Defaults` element, which specifies information used unless overridden for a particular application.

The following table lists the main elements allowed in `structapps.fm` as children of the `StructuredSetup` element. The table identifies the sections that discuss each of those elements and the elements they may contain.

Element	Discussed in
`ApplicationName`	"Defining an application" on page 53
`CharacterEncoding`	"Specifying the character encoding for SGML files" on page 66
`CssPreferences`	"Managing CSS generation" on page 62
`ConditionalText`	"Modifications to the default translation" on page 344 in Chapter 23, "Translating Conditional Text."
`Defaults`	"Providing default information" on page 54
`DOCTYPE`	"Specifying a document element" on page 55

Element	Discussed in
DTD	“Specifying a DTD” on page 56
Entities	“Specifying entities” on page 57
Entity	“Specifying the location of individual entities” on page 59
EntityCatalogFile	“Specifying entities through an entity catalog” on page 58
EntityName	“Specifying the location of individual entities” on page 59
EntitySearchPaths	“Specifying a search path for external entity files” on page 63
ExternalXRef	“Translating external cross-references to and from XML” on page 326
FileName	“Specifying the location of individual entities” on page 59
	“Specifying public identifiers” on page 61
FileExtensionOverride	“Specifying filename extensions” on page 56
FileNamePattern	“Specifying names for external entity files” on page 60
FrameDefaultAPIClient	“Specifying a structure API client” on page 66
MaxErrorMessages	“Limiting the length of a log file” on page 70
Namespace	“Enabling namespaces” on page 56
Path	“Specifying a search path for external entity files” on page 63
	“Specifying a search path for including files in rules documents” on page 64
Public	“Specifying public identifiers” on page 61
PublicId	“Specifying public identifiers” on page 61
ReadWriteRules	“Specifying a read/write rules document” on page 55
RulesSearchPaths	“Specifying a search path for including files in rules documents” on page 64
SGMLApplication	“Defining an application” on page 53
SGMLDeclaration	“Specifying an SGML declaration” on page 57
Template	“Specifying a FrameMaker template” on page 57
UseAPIClient	“Specifying a structure API client” on page 66
UseDefaultAPIClient	“Specifying a structure API client” on page 66
Graphic	“How FrameMaker searches filename patterns” on page 60
Mapping	“Specifying entities through an entity catalog” on page 58
Notation	“Specifying names for external entity files” on page 60
FmTranslator	“Specifying a structure API client” on page 66
XMLApplication	“Defining an application” on page 53

Element	Discussed in
XMLDisplayEncoding	"Specifying the character encoding for XML files" on page 68
XMLExportEncoding	"Specifying the character encoding for XML files" on page 68

Defining an application

FrameMaker collects all information pertaining to conversion of a related set of documents into an SGMLApplication or XMLApplication element. This element has one required child element and several optional child elements.

The first child of a parent SGMLApplication or XMLApplication element must be ApplicationName and gives the name of the application. It looks like:

Application Name: *name*

where *name* is a name to identify your application in the Set Structure Application and Use Structure Application dialog boxes. You cannot use the same name for multiple structure applications.

If present, the optional child elements can occur in any order and can include the following elements, discussed in the named sections:

Element	Discussed in
DOCTYPE	"Specifying a document element" on page 55
FileExtensionOverride	"Specifying filename extensions" on page 56
ReadWriteRules	"Specifying a read/write rules document" on page 55
DTD	"Specifying a DTD" on page 56
Template	"Specifying a FrameMaker template" on page 57
SGMLDeclaration	"Specifying an SGML declaration" on page 57
Entities	"Specifying entities" on page 57
RulesSearchPaths	"Specifying a search path for including files in rules documents" on page 64
UseAPIClient	"Specifying a structure API client" on page 66
UseDefaultAPIClient	"Specifying a structure API client" on page 66
CharacterEncoding	"Specifying the character encoding for XML files" on page 68

Several of these elements provide pathnames. If a relative pathname is given, the software looks for the file in several places (entities and read/write rules files only; hence RulesSearchPaths and EntitySearchPaths elements):

- The directory containing the file being processed. For example, if you're opening a DTD, the software first searches the directory in which it found the DTD only. If the pathname

is absolute, it looks there. If it can't find it via the specified path, the log reports an error and the operation is aborted.

- $STRUCTDIR (for information on what directory this is, see "Location of structure files" on page 49)
- The directory from which you started FrameMaker

The following is a typical application; note that these pathnames are for Unix systems:

Application name: ReportNumeric
DOCTYPE: Report
Read/write rules: reports/rules/repnumer.rul
DTD: reports/repnumer.dtd
Template: reports/repnumer
Rules search paths
1: $STRUCTDIR/reports/rules
Use default API client.

If an application definition includes any of these elements, the value in the application definition overrides any value for that element in the `Defaults` element. The sections following the next section describe these elements in detail.

Providing default information

Some of the information you provide for individual applications may be common to all your applications. For such information you can specify defaults that are used whenever an application doesn't provide its own version of the information. You use the `Defaults` element to provide such information.

If present, the optional child elements of `Defaults` can occur in any order (with the exception of the `Graphics` element, it must be the last child) and can include the following elements, discussed in the named sections:

Element	Discussed in
`ReadWriteRules`	"Specifying a read/write rules document" on page 55
`DTD`	"Specifying a DTD" on page 56
`Namespace`	"Enabling namespaces" on page 56
`Template`	"Specifying a FrameMaker template" on page 57
`SGMLDeclaration`	"Specifying an SGML declaration" on page 57
`Entities`	"Specifying entities" on page 57
`RulesSearchPaths`	"Specifying a search path for including files in rules documents" on page 64
`FrameDefaultAPIClient`	"Specifying a structure API client" on page 66
`UseAPIClient`	"Specifying a structure API client" on page 66
`MaxErrorMessages`	"Limiting the length of a log file" on page 70

Element	Discussed in
CharacterEncoding	"Specifying the character encoding for XML files" on page 68
Graphics	"How FrameMaker searches filename patterns" on page 60

Specifying a document element

The DOCTYPE element specifies the generic identifier of the DOCTYPE declaration and root element in markup documents used with this application. If you open a markup document with the matching document element specified in the DOCTYPE declaration, FrameMaker uses this application when translating the document. The element looks like:

DOCTYPE: *doctype*

where *doctype* identifies a document element.

For example,

DOCTYPE: chapter

matches a markup document with the following declaration:

```
<!DOCTYPE chapter ...>
```

If more than one application defined in the structapps.fm file specifies the same document element, and the end user opens a file with that document element, the software gives the user a choice of which of these applications to use. If the user opens a markup document for which no application specifies its document element, the software gives the user the choice of all defined applications.

You can use more than one DOCTYPE element for an application, if that application is applicable to multiple document elements. For example, if the Book application applies when the document element is either chapter or appendix, you can use this definition:

Application name: Book
DOCTYPE: chapter
DOCTYPE: appendix
...

The DOCTYPE element can be a child of a parent SGMLApplication or XMLApplication element only.

Specifying a read/write rules document

The ReadWriteRules element specifies the read/write rules document associated with the application. It looks like:

ReadWriteRules: *rules*

where *rules* is the pathname of a FrameMaker read/write rules document.

You can have only one ReadWriteRules element for each application. It can also be a child of the Defaults element.

Specifying a DTD

The DTD element specifies a file containing the external DTD subset the software uses when importing and exporting an markup document. It looks like:

DTD: *dtd*

where *dtd* is the pathname of a file containing a document type declaration subset.

Note that the file you specify with the DTD element must be an external DTD subset. It cannot be a complete DTD. That is, the file cannot have the form:

```
<!DOCTYPE book [
   <!element book . . .>
   . . .
]>
```

Instead, it should simply have the form:

```
<!element book . . .>
. . .
```

For more information on external DTD subsets, see "XML and SGML DTDs" on page 13.

You can have only one DTD element for each application. It can also be a child of the Defaults element.

Enabling namespaces

The Namespace element specifies whether the current XML structure application supports namespaces in the XML. This element can contain either Disable or Enable as a child element. The Namespace element looks like this:

Namespace
  Enable

You can have only one Namespace element for each XML structure application. It can also be a child of the Defaults element.

Specifying filename extensions

The FileExtensionOverride element specifies a filename extension to use when saving a FrameMaker document as XML. This is most useful when saving XHTML documents. Some web browsers that support XHTML can only read files with a .htm or .html extension. When you save a document as XML (even using the XHTML doctype) FrameMaker gives the file a .xml extension by default. You can use this element to specify

a .htm extension when saving a document as XHTML. The FileExtensionOverride element looks like this:

FileExtensionOverride: *extension*

where *extension* is the string for the filename extension, minus the dot character. You can have only one FileExtensionOverride element for each XML structure application.

Specifying a FrameMaker template

The Template element specifies the location of the FrameMaker template. It looks like:

Template: *template*

where *template* is the pathname of a FrameMaker template.

The software uses this template to create new FrameMaker documents from markup documents, which may be single documents resulting from the Open or Import command or documents in a book created through the Open command.

If this element is not present, the software creates new portrait documents as needed. When you import a markup document into an existing document, the software uses the import template only to access reference elements that are stored on the template's reference page. (For information about reference elements, see , "Translating SDATA entities as FrameMaker reference elements."

You can have only one Template element for each application. It can also be a child of the Defaults element.

Specifying an SGML declaration

The SGMLDeclaration element specifies the location of a file containing a valid SGML declaration. It is used only for SGML applications and is erroneous as a child of an XMLApplication element. The SGMLDeclaration element looks like:

SGMLDeclaration: *declaration*

where *declaration* is the pathname of the SGML declaration file.

You can have only one SGMLDeclaration element for each application. It can also be a child of the Defaults element.

Specifying entities

To specify the location of various entities, you use the Entities element. The possible child elements of a parent Entities element are:

Element	Discussed in
EntityCatalogFile	"Specifying entities through an entity catalog" on page 58
Entity	"Specifying the location of individual entities" on page 59
FileNamePattern	"Specifying names for external entity files" on page 60

Element	Discussed in
Public	"Specifying public identifiers" on page 61
EntitySearchPaths	"Specifying a search path for external entity files" on page 63

If you use the EntityCatalogFile element, you cannot use any of the elements Entity, FilenamePattern, or Public.

You can have only one Entities element for each application, although that Entities element can have more than one of some of its child elements. The Entities element can also be a child of the Defaults element.

Specifying entities through an entity catalog

The EntityCatalogFile element specifies a file containing mappings of an entity's public identifier or entity name to a filename. It looks like:

Entity locations
Entity catalog file: *fname*

where *fname* is the filename of the entity catalog. Entity catalogs and their specified format are described later.

You can specify multiple EntityCatalogFile elements in a single Entities element. If you use this element, you cannot use any of the Entity, FilenamePattern, or Public elements.

You can use the EntityCatalogFile element both in the Entities element of the Defaults element and in an SGMLApplication or XMLApplication element to specify information for a particular application. When searching for an external entity, FrameMaker searches the application's entity catalogs before searching those in default EntityCatalogFile elements.

If you have an EntityCatalogFile element in an application definition, the software ignores Entity, FilenamePattern, and Public elements in the Defaults element.

Why use entity catalogs

Technical Resolution 9401:1994 published by SGML Open relates to entity management issues affecting how SGML documents work with each other:

- Interpreting external identifiers in entity declarations so that an SGML document can be processed by different tools on a single computer system
- Moving SGML documents to different computers in a way that preserves the association of external identifiers in entity declarations with the correct files or other storage objects

The technical resolution uses *entity catalogs* and an interchange packaging scheme to address these issues. FrameMaker supports such entity catalogs with the EntityCatalogFile element.

Entity catalog format

Each entry in the entity catalog file associates a filename with information about an external entity that appears in a markup document. For example, the following are catalog entries that associate a public identifier with a filename:

```
PUBLIC "ISO 8879-1986//ENTITIES Added Latin 1//EN" "isolat1.ent"
PUBLIC "-//USA/AAP//DTD BK-1//EN" "aapbook.dtd"
```

In addition to entries mapping public identifiers to filenames, an entry can associate an entity name with a filename:

```
ENTITY "chips" "graphics\chips.tif"
```

A single catalog can contain both types of entry.

If the specified filename in a catalog entry is a relative pathname, the path is relative to the location of the catalog entry file.

For a complete description of the syntax of a catalog entry, see *Technical Resolution 9401:1994 Entity Management* published by SGML Open.

How FrameMaker searches entity catalogs

There may be multiple catalog files for a single application. When trying to locate a particular external entity, FrameMaker searches the files one at a time until it finds the entry it is looking for. In each file, the software first searches for an entity using the external entity's public identifier. If the software finds the identifier, it uses the associated filename to locate the entity. If it does not find the public identifier, the software searches the file looking for the entity name. If it does not find the entity name either, the software continues searching in the next catalog file.

In some circumstances, a system identifier specified in an external entity declaration may not be valid. If so, FrameMaker uses public identifier and entity name mappings.

Specifying the location of individual entities

Instead of using an entity catalog to associate entities with files, you can use the `Entity` element as a child of a parent `Entities` element. This element allows you to directly associate a filename with an individual entity. It looks like:

Entity locations
Entity name: *ename*
Filename: *fname*

where *ename* is the name of an entity and *fname* is a filename.

You can specify multiple child `Entity` elements for a single `Entities` element. You use the `FilenamePattern` and `EntitySearchPaths` elements to help the software find these files.

The `Entity` element can be a child of a parent `Entities` element in the `Defaults` element to set default entity information and of a parent `SGMLApplication` or

XMLApplication element to specify information for a particular application. When searching for an external entity, the software searches the application's entity locations before searching those in the default Entity elements.

Specifying names for external entity files

One or more FilenamePattern elements can appear as a child of a parent Entities element to tell the software how to locate an external entity.

A FilenamePattern element does not apply to an entity for which there is an Entity element. Otherwise, it applies to all external entities except those with an external identifier that includes a public identifier but no system identifier. The FilenamePattern looks like:

Entity locations:
FilenamePattern: *pattern*

where *pattern* is a string representing a device-dependent filename. The three variables that can appear within *pattern* are interpreted as follows:

Variable	Interpretation
$(System)	The system identifier from the entity declaration
$(Notation)	The notation name from the entity declaration of an external data entity
$(Entity)	The entity name

Case is not significant in variable names, although it may be significant in the values of the variables. If a variable is undefined in a particular context, that variable evaluates to the empty string.

There can be multiple child FilenamePattern elements in a parent Entities element. The software assumes the last pattern in the Entities element is:

FilenamePattern: $(System)

Thus, if no FilenamePattern elements appear or even if no Entities element appears, the software assumes system identifiers are complete pathnames and will check search paths to locate the file.

How FrameMaker searches filename patterns

When locating an external entity, FrameMaker tests the value of the *pattern* arguments in successive FilenamePattern elements that have the same parent Entities element, in the order they occur, until it finds the name of an existing file. As it tests each *pattern*, it substitutes relevant information from the entity's declaration for variables in *pattern*.

You can use the FilenamePattern element both in the Entities element of the Defaults element and in an SGMLApplication element to specify information for a particular application. When searching for an external entity, FrameMaker tests all the filename patterns specified for the application before it tests those in default FilenamePattern elements.

Example
Suppose the Entities element looks like:

Entity locations:
FilenamePattern: $(System).sgm
FilenamePattern: $(System).$(Notation)

and the markup document contains:

```
<!ENTITY intro SYSTEM "introduction.xml">
<!ENTITY chips SYSTEM "chipsfile" NDATA cgm>
. . .
&intro;
. . .
<graphic entity=chips>
```

When processing the reference to intro, the software searches for a file called introduction.xml. It is an error if the file does not exist.

When processing the entity attribute of the graphic element, FrameMaker searches for a file named chipsfile.cgm. If one is not found, it then looks for chipsfile.CGM, assuming that the NAMECASE GENERAL parameter of the associated SGML declaration is NAMECASE GENERAL YES.

Important: The NAMECASE GENERAL parameter of the SGML declaration determines the case-sensitivity of notation names. For XML, the implied setting for this parameter is NO, which means that names are case-sensitive.

For SGML, the value of this parameter in the reference concrete syntax is NAMECASE GENERAL YES. With this declaration, the SGML parser forces notation names to uppercase. Because of this, and because the UNIX file system is case-sensitive, on a UNIX system, if you have the default SGML declaration and you use the $(notation) variable with the filename element, that portion of the name must be uppercase.

Specifying public identifiers

The Public element of an Entities element tells the software how to process an external identifier that has a public identifier but no system identifier. It looks like:

Entity locations:
Public ID: *pid*
Filename: *fname*

where *pid* is a public identifier and *fname* is the name of a file to be associated with the entity using the public identifier.

You can give multiple Public elements in the same parent Entities element. If you want to give multiple filenames to search for a particular public identifier, you can specify the same public identifier in multiple Public elements.

You can use the Public element both in the Entities element of the Defaults element and in an Entities element of an SGMLApplication or XMLApplication element to specify information for a particular application. If a Public element occurs as a child of an SGMLApplication or XMLApplication element, that identifier is used in preference to one occurring as a child of the Defaults element.

Managing CSS generation

The Stylesheets element of an XML structure application tells the software how to treat the use of stylesheets for a given XML document. You can specify whether to use an existing stylesheet, or whether FrameMaker should generate a new one and use that for the exported XML. This element also contains instructions concerning the use of attributes and stylesheet processing instructions. The Stylesheets element looks like:

Stylesheets:
CssPreferences:
Generate CSS2: *enable/disable*
AddFmCSSAttrToXML: *enable/disable*
RetainStylesheetPIs: *enable/disable*
XmlStylesheet:
Type: *stylesheet_type*
URI: *path*

An XML application can have only one Stylesheets element. It can also be a child of the Defaults element. You can specify any number of stylesheets, and the exported XML will include references to each one.

How the Stylesheets elements CSS generation

When you save a document to XML, FrameMaker can either use an existing stylesheet, or generate a new one from the current EDD. How FrameMaker generates a stylesheet is determined by the values of the children of the Stylesheet element. For more information about how FrameMaker converts EDD information into a stylesheet, see Chapter 15, "Saving EDD Formatting Information as a CSS Stylesheet."

GenerateCSS2 Specifies whether FrameMaker will generate a CSS when you save the document as XML. It can be set to *Enable* or *Disable*. When this is set to Enable, FrameMaker generates a CSS. If a path is provided in StylesheetURI, FrameMaker saves the stylesheet to that location, with that filename. Otherwise, it saves the stylesheet to the same location as the XML document with a filename *xmldoc*.css, where *xmldoc* is the name of the XML document you're saving.

AddFmCSSAttrToXml Specifies whether FrameMaker will write instances of the *fmcssattr* attribute to elements in the XML document. It can be set to *Enable* or *Disable*. An EDD can include context selectors as criteria to assign format rules. CSS has no equivalent to this. When this is set to *Enable*, FrameMaker uses the *fmcssattribute* in certain elements so the CSS can achieve the same formatting as the EDD.

RetainStylesheetPIs Specifies whether FrameMaker will retain the stylesheet declaration for import and export of XML. It can be set to *Enable* or *Disable*. When this is set to *Enable*, FrameMaker does the following:

- On import, it stores the XML document's stylesheet PI as a marker in the FrameMaker document.
- On export, it writes the content of stylesheet PI marker in the resulting XML document.

StylesheetType Specifies the type of stylesheet. It contains a string for the stylesheet type. Currently, you can specify `CSS` (upper or lower case) or `XLS` (upper or lower case). If you specify `XLS`, FrameMaker will not generate a stylesheet.

StylesheetURI Specifies the URI for the stylesheet. It contains a string; for example, `/$STRUCTDIR/xml/xhtml/app/xhtml.css`.

Specifying a search path for external entity files

The `EntitySearchPaths` child of a parent `Entities` element tells the software what directories to search for the files indicated by `Entity`, `FilenamePattern`, and `Public` elements. It looks like:

Entity locations:
EntitySearchPaths
1: *directory*$_1$
...
N: *directory*$_n$

where each *directory*$_i$ is a device-dependent directory name. The three variables and their abbreviations that can be used to specify a directory are as follows:

Variable	Abbreviation	Interpretation
`$HOME`	`~`	The user's home directory
`$SRCDIR`	`.`	The directory containing the document entity being processed
`$STRUCTDIR`		The structure directory in use (for information on what directory this is, see "Location of structure files" on page 49)

Each *directory*$_i$ value can be an absolute pathname or relative to `$SRCDIR`.

How FrameMaker searches for entity files

To locate an external entity, FrameMaker searches the specified directories in the order listed. You can use the `EntitySearchPaths` element both in the `Entities` element of the `Defaults` element and in an `XMLApplication` or `SGMLApplication` element. When searching for an external entity, FrameMaker searches the directories named in the `EntitySearchPaths` element for the application before it searches those in a default `EntitySearchPaths` element.

An Entities element can contain only one EntitySearchPaths element. The software assumes the EntitySearchPaths element ends this way:

EntitySearchPaths
. . .
N: $SRCDIR

Thus, if there is no EntitySearchPaths element, the software assumes all markup files are in the same directory.

Example
Assume the Defaults element is defined as follows:

Defaults
Entity locations:
FilenamePattern: $(System).sgm
FilenamePattern: $(System).$(Notation)
EntitySearchPaths
1: $HOME
2: $SRCDIR

and the markup document contains:

```
<!ENTITY intro SYSTEM "introduction.xml">
<!ENTITY chips SYSTEM "chipsfile" NDATA cgm>
. . .
&intro;
. . .
<graphic entity=chips>
```

When processing the reference to intro, the software looks for the files:

```
$HOME/introduction.xml
$SRCDIR/introduction.xml
```

until it finds one of those files. When processing the graphic element, the software searches in order for:

```
$HOME/chipsfile.cgm
$SRCDIR/chipsfile.cgm
```

Specifying a search path for including files in rules documents

The RulesSearchPaths element is analogous to the EntitySearchPaths element, but it pertains to additional files you include in a read/write rules document rather than to

external entities referenced within a markup document. Its `Path` child elements indicate individual directories. It looks like:

RulesSearchPaths:
1: *directory*$_1$
. . .
N: *directory*$_n$

where each *directory*$_i$ is a device-dependent directory name. The three variables and their abbreviations that can be used to specify a directory are as follows:

Variable	Abbreviation	Interpretation
$HOME	~	The user's home directory
$RULESDIR	.	The directory containing the current read/write rules document (for Macintosh and Unix, only)
$STRUCTDIR		The structure directory in use (for information on what directory this is, see "Location of structure files" on page 49)

Each *directory*$_i$ value can be an absolute pathname or relative to $RULESDIR.

How FrameMaker searches for rules files

Only one RulesSearchPaths element can occur as the child of a single parent XMLApplication or SGMLApplication element or parent Defaults element. When searching for a file you include in an read/write rules document, FrameMaker searches the directories named in the RulesSearchPaths element for the application before it searches those in the RulesSearchPaths element of the Defaults element.

The software assumes RulesSearchPaths ends in this way:

RulesSearchpaths:

. . .

N: $RULESDIR

Thus, if there is no RulesSearchPaths element, the software assumes all files you include in the read/write rules document are in the same directory as your rules document.

Specifying a structure API client

In an application definition, the UseDefaultAPIClient element tells the software that your application does not use a special client for markup translation. In the defaults section, the FrameDefaultAPIClient element serves the same purpose. The default client is named FmTranslator.

If you do need a structure API client, use the UseAPIClient element in either context. For information on creating structure API clients for a structure application, see the online manual *Structure Import/Export API Programmer's Guide*.

Specifying the character encoding for SGML files

The CharacterEncoding element tells the software which encoding to use for the SGML text. Typically, this is only important on non-Western systems, or in SGML applications that encounter SGML files using double-byte text. It can contain one of the following child elements: ISOLatin1, ASCII, ANSI, MacASCII, ShiftJIS, KSC8EUC, GB8EUC, CNSEUC, Big5, JIS8EUC. The CharacterEncoding element looks like this:

CharacterEncoding:

ShiftJIS

On a non-Western system, the text for an SGML file can contain double-byte text. This text can be in any one of a number of different text encodings. For example, Japanese text on a Unix system is typically in EUC, while on the Macintosh it is typically in Shift-JIS.

FrameMaker can interpret SGML files that contain double-byte text in *#PCDATA*, *RCDATA*, and *CDATA*. The software expects all other text to be within the 7-bit ASCII range (which is supported by all Asian fonts). This means that document content can be in double-byte encodings, but the markup must be in the ASCII range. Typically, for example, the only text in a DTD that will contain double-byte characters would be text used to specify attribute values.

> **Important:** For SGML documents, you should not use accented characters in element tag names nor attribute names. If you use such characters, FrameMaker may not be able to correctly import or export the document.

To import and export SGML that contains double-byte text, you should specify the character encoding to use, either as a default for all applications, or for a specific SGML application. For a given SGML application there can only be one encoding. If you don't specify an encoding for your application, FrameMaker determines the encoding to use by considering the current default user interface language and the current operating system; for the current language, it uses the operating system's default encoding. The default encodings for the supported operating systems are:

	Windows 95/NT	**Macintosh**	**Unix**
Roman languages	ANSI	Macintosh ASCII	ISOLatin-1
Japanese	Shift-JIS	Shift-JIS	JIS8 EUC
Simplified Chinese	GB8 EUC	GB8 EUC	GB8 EUC
Traditional Chinese	Big5	Big5	CNS EUC
Korean	KSC8 EUC	KSC8 EUC	KSC8 EUC

It is possible to have an Asian language for the user interface, but the content of the document files in Roman fonts. In this case, any exported Roman text that falls outside of the ASCII range will be garbled. For this reason, we recommend that you specify an encoding for any application that might be used on a non-Western system.

The template for your application must use fonts that support the language implied by the encoding you specify. Otherwise, the text will appear garbled when imported into the template. You can fix this problem after the fact by specifying different fonts to use in the resulting files.

Specifying the character encoding for XML files

The XML specification supports UNICODE characters for document content and markup tokens. In XML the given encoding is specified in the document prolog. The following example shows a specification for ShiftJIS character encoding:

```
<?xml version="1.0" encoding="Shift_JIS" ?>
```

The XML specification states that an XML document must either specify an encoding in the prolog, or it must be UTF-8 or UTF-16. FrameMaker follows this specification by assuming UTF-8 by default if there is no encoding specified in the XML file.

If you read an XML file with character encoding that doesn't match either the declared encoding or the default encoding (if no encoding is declared), it is likely that the import process will encounter a character that doesn't match the encoding FrameMaker uses. In that case, you will get a parsing error that says the document is not well-formed due to a bad token.

FrameMaker uses the encoding statement in the document prolog to determine which encoding to use. The statement must specify one of the encodings supported by your specific FrameMaker installation. FrameMaker ships with support for the following encodings:

Big5	KSC_5601
EUC-CN	macintosh
EUC-JP	Shift_JIS
EUC-KR	US-ASCII
EUC-TW	UTF-16
GB2312	UTF-8
ISO-8859-1	windows-1252

You can add other encodings to your FrameMaker installation—see "Unicode and character encodings" on page 27.

FrameMaker converts the encoding of the XML document to an internal display encoding. In this way FrameMaker fully supports Unicode characters for text that is in *#PCDATA*, *RCDATA*, and *CDATA*. For any #PCDATA character that it cannot interpret, FrameMaker uses a marker of type UNKNOWNCHAR to represent the character. For unknown CDATA characters, FrameMaker uses XML character references.

The following sections describe how to control the display encoding that FrameMaker uses, and how to specify an encoding when you save a document as XML.

Display encoding

On import, FrameMaker converts the XML encoding to a display encoding that is appropriate for a given language. However, FrameMaker cannot automatically determine which conversion to make. Although the XML document prolog specifies an encoding, the document may contain elements or other constructs that override the language implied by

that encoding. As a result, you should specify a display encoding for the structure application. The display encodings you can specify are:

Display encoding:	For this language:
FrameRoman	Western European languages
JISX0208.ShiftJIS	Japanese
BIG5	Traditional Chinese
GB2312-80.EUC	Simplified Chinese
KSC5601-1992	Korean

By default, FrameMaker uses the display encoding that matches the locale of your operating system. To specify a different display encoding, use the XmlDisplayEncoding element. XmlDisplayEncoding can contain one child element to specify one of the supported display encodings.

The display encoding also determines how FrameMaker interprets the characters in markup tokens such as GIs and attribute names. If FrameMaker encounters such a token with an unknown character, FrameMaker drops the token. For more information, see "Supported characters in element and attribute names" on page 26.

For example, if your operating system locale is French, German, or English FrameMaker uses FrameRoman by default. This is true, even if the XML prolog specifies an encoding for a different language, such as ShiftJIS. To import XML encoded as ShiftJIS, you would use the XmlDisplayEncoding element to specify JISX0208.ShiftJIS, as follows:

XmlDisplayEncoding:
JISX0208.ShiftJIS

When you specify such an encoding, FrameMaker uses that encoding as the default for all the *#PCDATA*, *RCDATA*, and *CDATA* in the imported XML. Markup tokens that include characters in the upper range of the display encoding are interpreted correctly. If you have fonts installed for the display encoding, then the text will appear as intended.

For another example, assume you have a version of US English FrameMaker installed on a Traditional Chinese operating system. By default, FrameMaker uses Big5 as the display encoding. It also supports any Big5 characters that are used in GIs and attribute names. If you are importing an XML document that is in English, you would need to specify FrameRoman as the display encoding.

Note that XML the XML standard includes the xml:lang attribute. This attribute can specify a change of language for an element and its content. If that language is one of those listed in the table of display encodings, a change made by this attribute take precedence over the setting made via XmlDisplayEncoding.

Finally, the template for your application must use fonts that support the given language. Otherwise, the text will appear garbled when imported into the template. You can fix this problem by specifying different fonts to use in the resulting files.

Exporting XML

Your XML structure application can include an `XmlExportEncoding` element to specify the encoding to use when you save a document as XML. FrameMaker determines which encoding to use according to the following:

If:	FrameMaker uses:
1. The structure application specifies a value for XmlExportEncoding, and that encoding is supported	The specified encoding
2. 1 is not true, and the original XML source specified an encoding, and that encoding is supported	The encoding that was specified in the original XML source
3. 1 and 2 are not true	UTF-8

The XmlExportEncoding element contains a string for the name of an encoding. The name you provide must conform with the IANA naming conventions. The standard installation of FrameMaker supports the encodings that are listed at the beginning of this discussion (see page 68).

For example, to export your document as ISOLatin1, use the `XmlExportEncoding` element as follows:

XmlExportEncoding:
ISO-8859-1

Limiting the length of a log file

The `MaxErrorMessages` child element of the `Defaults` element allows you to limit the length of structure error reports. It looks like:

MaxErrorMessages: *n*

where *n* is the desired limit. If *n* is less than 10, the software resets it to 10. This must be the last child of the parent `Defaults` element.

By default, FrameMaker does not write more than 150 messages (error messages and warnings) to a single log file.

Messages pertaining to opening and closing book components are not included in this limit. Messages generated through your own structure API client are also not counted, although if you wish, you can count them using your own code.

In documents that generate large numbers of messages, the 151st message is replaced with a note that additional messages have been suppressed.

Note that processing continues, even though further messages are not reported. This message limit is reset for every file processed and for each component of a book.

Log files

FrameMaker log files give you information used to identify and correct errors in your application.

Generating log files

FrameMaker can produce a log file of errors and warnings for each file it processes. If it runs without encountering errors or other conditions requiring messages, it does not create a log file. The one exception to this is that the UNIX batch utilities always produce a log file so that you know what files were processed.

FrameMaker generates messages for conditions such as:

- Markup syntax errors
- Read/write rule syntax errors
- Missing or otherwise unavailable files
- Missing entities
- Inconsistencies between read/write rules and the relevant EDD or DTD

Messages in a log file

Messages written to the log file include warnings, errors, and fatal errors.

Warnings are notifications of potential problems; a warning does not necessarily mean something is wrong with the input. For example, when the software creates an EDD from a DTD, it issues a warning if it encounters an `element` rule in a read/write rules document for a generic identifier that doesn't exist in the DTD. This situation is legal, but the software warns you in case you have misspelled the generic identifier in the rules document. For all warnings, the software writes a message and continues processing.

Errors indicate actual problems in the processing. For example, when FrameMaker updates an existing EDD, it reports an error for a FrameMaker element definition that has no counterpart in the DTD. For most errors, the software writes a message and continues processing.

The third category of message indicates *fatal errors*. These errors cause processing to stop. For example, if FrameMaker encounters a syntax error in a rules document, it stops processing.

To aid in debugging your structure application, FrameMaker provides a facility for locating certain problems. Choose Check Read/Write Rules from the Developer Tools menu. FrameMaker uses the current application to check the validity of the rules document in that application. The command can find many potential problems and most fatal errors.

Using hypertext links

When FrameMaker creates a log file, where possible it includes hypertext links as an additional debugging tool. Each message in a log file has one of the forms shown in the following examples:

```
-> /usr/vpg/ch1.fm;
   Invalid property specified for element "AFrame".
```

or

```
/usr/vpg/ch1.sgm; line 25;
Required attribute "lang" is missing for element "list".
```

The first part of the message indicates the location of the problem in the source document; it always includes the source document's filename. If the source is a FrameMaker document, such as a rules document or a FrameMaker document being imported or exported, there is an arrow to the left as in the first example. The arrow indicates a hypertext link. If you activate the link, the software opens the document to the page containing the problem. If the source isn't a FrameMaker document, there is no hypertext link. In this situation, the description includes the line number in the file, as in the second example. You must open the file in an appropriate editor.

The second part of the message describes the particular problem encountered. This part of the message always contains a hypertext link to an explanation of the message.

Setting the length of a log file

Processing documents can produce extremely large log files. In practice, you are unlikely to look at all pages of an extremely large log file. For this reason and to save space and time, FrameMaker limits the number of messages it sends to the log file for a given document. For information on how to change that number, see "Limiting the length of a log file" on page 70.

Other special files

There are several other file types you must work with in creating a structure application. These files are discussed in other chapters. For information on creating an EDD, see Part II, "Working with an EDD." For information on creating read/write rules files, see Chapter 14, "Read/Write Rules and Their Syntax." For information on ISO public entity set files, see Appendix F, "ISO Public Entities."

Part II Working with an EDD

Part II explains how to develop an element definition document (EDD) and define elements in it. If you're developing an EDD as part of a larger structure application, you should be familiar with Part I, "Developing a FrameMaker structure application," before using the material in this part.

The chapters in this part are:

- Chapter 8, "Developing an Element Definition Document (EDD)"

  Discusses the process of developing an EDD—from creating a new EDD through building a structured template from your element definitions. Read this chapter first for an overall understanding of the process. The chapter also has a list of all elements in an EDD's Element Catalog, with links to sections where the elements are covered in the syntax chapters.

- Chapter 9, "Structure Rules for Containers, Tables, and Footnotes"
  Chapter 11, "Attribute Definitions"
  Chapter 10, "Text Format Rules for Containers, Tables, and Footnotes"
  Chapter 12, "Object Format Rules"

  The four syntax chapters describe structure rules, attribute definitions, text format rules, and object format rules, and show examples of these constructs in element definitions.

8 Developing an Element Definition Document (EDD)

An *element definition document* (EDD) contains the structure rules, attribute definitions, and format rules for all of the elements in a group of FrameMaker documents. You write and maintain the definitions in the EDD and then convert them to an Element Catalog in a structured template. To work with the elements, end users create documents from the template or exporting from, then importing into a new document.

You can start a new EDD in two ways. If you have a DTD that your end users' documents will follow, you can begin with the DTD and create an EDD with element definitions that correspond to constructs in the DTD. If you do not have a DTD, or if you do not plan to translate structured documents to markup, you can create an EDD and write its definitions entirely in FrameMaker.

An EDD is a regular structured FrameMaker document—it has an Element Catalog already set up with everything you need to define elements for end-users' documents. When developing the EDD, you insert elements from the catalog and in most cases provide values to fill out the definitions. For general information on working in structured documents, see the *FrameMaker User Guide*.

In this chapter

This chapter explains the process of developing an EDD and provides a summary of the elements in the EDD's Element Catalog. In the outline below, click a topic to go to its page.

A first look at the steps for developing an EDD:

- "Overview of the development process" on page 76

How to start or update an EDD:

- "Creating or updating an EDD from a DTD" on page 77
- "Starting an EDD without using a DTD" on page 80

Summary of the elements you work with in an EDD:

- "The Element Catalog in an EDD" on page 81

How to define elements and supply other information about them:

- "Defining preliminary settings in an EDD" on page 88
- "Organizing and commenting an EDD" on page 90
- "Writing element definitions" on page 91
- "Keyboard shortcuts for working in an EDD" on page 102

What to do when you're finished developing an EDD:

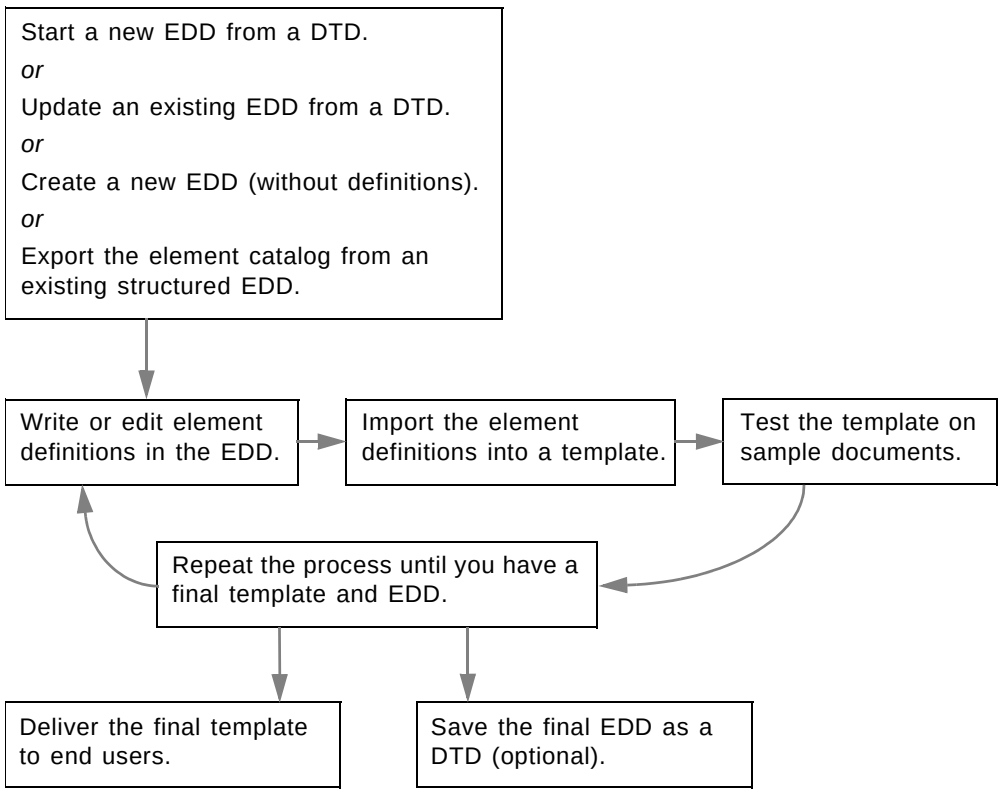
- "Creating an Element Catalog in a template" on page 103
- "Saving an EDD as a DTD for export" on page 105

Where to find structured documents and EDDs to review:

- "Sample documents and EDDs" on page 107

Overview of the development process

These are the basic steps you go through to create an EDD and to develop it as part of a workable template for your end users:



If you start the process with a DTD, FrameMaker creates a new EDD with definitions that correspond to the declarations in the DTD. In most cases, as part of this conversion you need to analyze the EDD and DTD together, develop read/write rules, and repeat the process until the translation is complete. You'll probably at least need to add format rules to the definitions in the EDD.

If you do not have a DTD, you can either export the Element Catalog from an existing FrameMaker document to create a new EDD or create a new empty EDD by selecting File>Developer Tools>New EDD. Then you either edit the definitions of the exported elements or write new element definitions from scratch.

When you're finished writing and editing element definitions in the EDD, you import the definitions into a FrameMaker template that also stores formats which may be referenced in your format rules. (You may need to coordinate this part of the process with a template designer responsible for formatting information.) Then you test the template on sample XML or SGML documents, revise the definitions in the EDD, reimport the definitions into the template, and repeat the process as necessary until you have a template that works the way you want it to.

Finally, you deliver the structured template to end users—in many cases, along with other pieces of a larger XML or SGML application. If you didn't start from a DTD but your users will export documents to markup data, you also need to save the EDD as a DTD.

At the end of the process, even though you are finished developing the EDD, you should normally keep it as a separate document to use for maintaining the element definitions.

Creating or updating an EDD from a DTD

If your documents need to conform to a DTD, you can use the DTD as a starting point for your EDD. FrameMaker creates a new EDD with element and attribute definitions that correspond to element declarations and attribute definition list declarations in the DTD. If the declarations in the DTD change, you can update the EDD to match.

About the DTD

A DTD is required in the prolog of an SGML document, and may be in the prolog of an XML document. The DTD provides element and attribute definition list declarations. The markup document can contain these declarations directly, or it can have an identifier that references a set of declarations stored separately in an external entity. This entity is sometimes called an *external DTD subset*.

If you start an EDD from a DTD, the DTD you use can be either a complete DTD at the beginning of a markup document or an external DTD subset stored in a separate file. In this section, the term *DTD* can mean either case.

For more information on DTDs and how they can be stored, see "XML and SGML DTDs" on page 13.

Read/write rules and the new EDD

When starting from a DTD, we recommend that you first create an initial EDD with no *read/write rules*—or with only a subset of the rules if you have some already developed. This lets you see how FrameMaker translates the DTD with little or no help from your rules.

Once you have both a DTD and an EDD, you can refine the translation in an iterative process of developing read/write rules. First analyze the DTD and new EDD together to plan how to modify the translation with rules. Then develop at least some of your rules, update the EDD from the DTD using the rules (see "Updating an EDD from a DTD" on page 79), test the results on sample markup documents, and repeat the process as many times as necessary. You may find it easiest to write and test only a few rules during each iteration. For a more detailed discussion of this process, see "Task 3. Creating read/write rules" on page 35.

You develop read/write rules in a special rules document that is part of a structure application. When you create an EDD from a DTD, you can specify which application (and hence which set of rules) to use with the EDD. For information on developing a read/write rules document, see Chapter 14, "Read/Write Rules and Their Syntax."

An application definition file (such as `structapps.fm`) describes what files are used in each structure application you deliver to an end user. If necessary, update an application definition in this file so that the application uses the appropriate read/write rules document. To do this, insert a `ReadWriteRules` element in the definition and type the pathname of the document. For more information, see "Application definition file" on page 50.

Creating an EDD from a DTD

To create an EDD from a DTD, choose Open DTD from the File>Structure Tools submenu in any open FrameMaker document. Select the DTD in the Open DTD dialog box, and then select an application in the Use Application dialog box.

The name of the application you select is stored in an `SGMLApplication` or `XMLApplication` element in the EDD for future updates and exports. All FrameMaker documents with an Element Catalog derived from the EDD use the application by default.

If you are creating an initial EDD without read/write rules, select <No Application>. This specifies a default structure application with no rules. When you open a DTD with <No Application>, FrameMaker displays a dialog box with the choice to specify whether the EDD will be for an XML or an SGML To specify a structure application for the EDD later, update the EDD from the DTD (see "Updating an EDD from a DTD" on page 79), and this time select the application, or insert and fill in an `SGMLApplication` or `XMLApplication` element in the EDD. For information on filling in the element, see "Setting a structure application" on page 89.

What happens during translation

A DTD has element and attribute definition list declarations that correspond to element and attribute definitions in an EDD. When you translate a DTD to an EDD, FrameMaker makes assumptions about how the constructs from the DTD should be represented. FrameMaker reads through the entire DTD, processing elements and their attributes one at a time. In the absence of read/write rules, the software translates an element declaration from the DTD

to a FrameMaker element definition of the same name, and it produces an attribute definition for each attribute defined for the element.

Note that DTDs contain syntactic information about the structure of a class of documents, but they do not address the semantics of elements they define. For example, DTDs do not distinguish between an element used to define an equation and one used to define a marker. For this reason, the default translation may not convert all of the markup elements correctly. (An exception to this is *CALS* tables. If your DTD uses the CALS table model, FrameMaker does recognize those elements as table elements.) You can modify the default translation using read/write rules.

For details on the translation of each type of element, see Part III, "Translating between markup data and FrameMaker."

Updating an EDD from a DTD

These are two of the reasons you may need to update an EDD:

- If you started the EDD from a DTD, in most cases you need to modify the translation by developing and testing read/write rules in an iterative process. As part of each iteration, you update the EDD using the DTD and your current set of read/write rules.
- If any element or attribute declarations in the DTD change, you update the EDD to revise the corresponding definitions in the EDD.

To update an EDD from a DTD, choose Import DTD from the File>Structure Tools submenu in the EDD. Select the DTD in the Import DTD dialog box. If the Use Application dialog box appears, select a structure application for the EDD. (Use Application appears only if no application is specified in the EDD.)

In the updated EDD, FrameMaker adds definitions for new elements from the DTD, removes definitions for elements that are no longer defined, and revises the content rules and attribute definitions for the remaining elements to match changes in the DTD and the current read/write rules. Any format rules and comments in the EDD are not affected, except for those in definitions that have been removed. (The software records these changes in a log file.) You can save the modified EDD if you want to keep the changes.

Log files for a translated DTD

If FrameMaker encounters any problems while starting or updating an EDD from a DTD, it produces a log file of warnings and errors. A warning is a notification of a potential problem, but it does not necessarily mean something is wrong with the DTD or the resulting EDD. An error indicates an actual problem in the processing; some errors can cause the processing to stop altogether.

A log file can have warning messages for conditions such as name changes, and it can have error messages for markup syntax errors, read/write rule errors, and missing files. If you're updating an EDD from a DTD, the log file also includes a list of changes made to

the EDD and may include error messages for inconsistencies between the DTD and the EDD.

This is an example of a message in a log file:

```
/usr/struct/tutorial/chapter.dtd; line 63
Parameter entity name longer than (NAMELEN-1); truncated
```

The first line in the message gives the location of the problem in the DTD. The second line describes the problem; you can click this line to see a longer explanation.

A log file is initially locked so that you can click in it to use the hypertext links. If you want to save the log file, you must first unlock it by pressing Esc Flk. (Press Esc Flk again to relock the file.) For general information on FrameMaker log files, see "Log files" on page 71.

Starting an EDD without using a DTD

If you do not have a DTD, or if you do not plan to translate structured documents to markup, you can start an EDD without using a DTD. You either create a new EDD and define the elements from scratch or create an EDD that has the element definitions from an existing FrameMaker document. When you start an EDD without a DTD, you can still save the EDD as a DTD later.

If you plan to translate documents to XML or SGML but do not yet have a DTD, we recommend that you begin with an EDD and then continue the development process from there. An EDD has richer semantics than a DTD and a set of tools that facilitate defining elements, so you will probably find it easier to develop an environment for your end users in an EDD.

Creating a new EDD

You can create a new EDD and enter all of the element definitions from scratch. The new EDD has a highest-level element called `ElementCatalog`, a `Version` element with the number of the current FrameMaker release, and one empty `Element` element.

To create a new EDD, choose New EDD from the File>Developer Tools submenu.

Exporting an Element Catalog to a new EDD

You can export the Element Catalog of an existing structured document to create an EDD. The new EDD has:

- a highest-level element called `ElementCatalog`
- a `Version` element with the number of the current FrameMaker release
- a `Para` element with the name of the structured document and the current date and time
- element definitions for all of the elements from the document's catalog

Exporting an Element Catalog is helpful when you already have a structured document that you'd like to use as a basis for other documents. You will probably need to add or edit element definitions in the new EDD.

To export an Element Catalog to a new EDD, choose Export Element Catalog as EDD from the File>Developer Tools menu in the structured document with the Element Catalog.

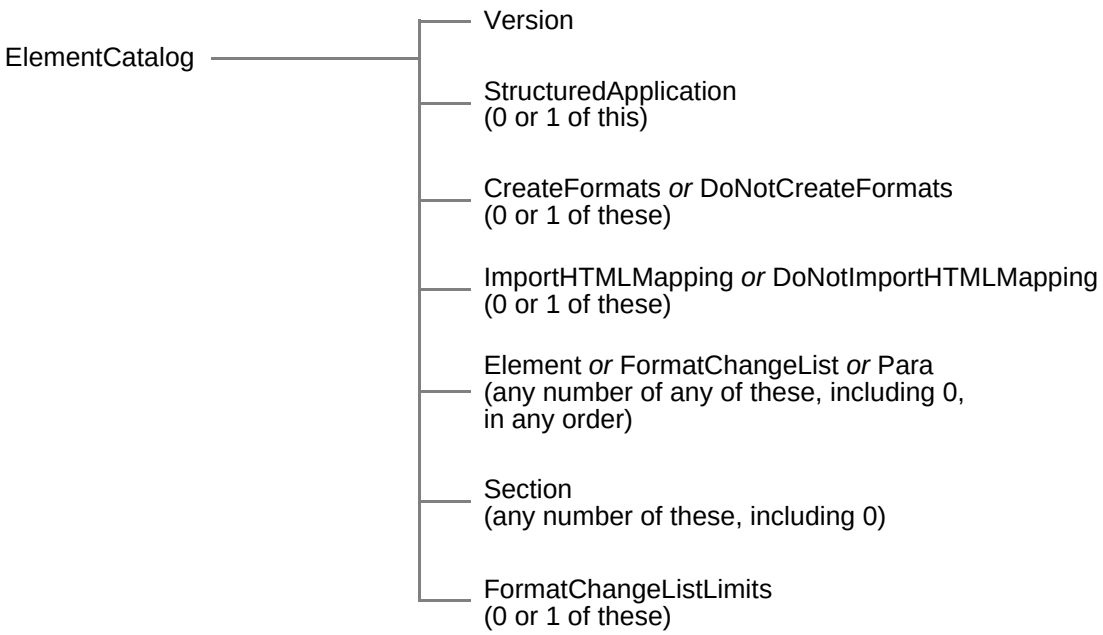
The exported EDD and the EDD from which it was created are equivalent in that they have the same element definitions. The two EDDs may differ, however, in the order and grouping of definitions. `Section`, `Head`, and `Para` elements from the original EDD are also not preserved in the exported EDD.

The Element Catalog in an EDD

An EDD is a regular structured FrameMaker document. It comes with an Element Catalog that has everything you need to define elements and to specify related information for your end users' documents. You insert elements from the catalog as you do in any other structured document, and in most cases provide values to fill out the rules.

High-level elements

The highest-level element in an EDD is `ElementCatalog`. It can have the following child elements, in the order shown. Only the `Version` element is required. `ElementCatalog` and the required `Version` are inserted automatically, along with the optional `CreateFormats` and `Element`, when you start a new EDD.



The Version element displays the number of the FrameMaker release used when the EDD was created; the number is not editable. The following list describes what the optional child elements are for and where you can find more discussion of them in this manual:

- StructuredApplication: Specifies an SGML or XML application for the EDD and for documents that use the EDD. You need to type the name of the application. See "Setting a structure application" on page 89
- CreateFormats or DoNotCreateFormats: Specifies whether or not to create formats automatically when you import element definitions into a template or other document. CreateFormats is inserted automatically in a new EDD. See "Specifying whether to create formats automatically" on page 89.
- ImportHTMLMapping or DoNotImportHTMLMapping: Specifies whether or not to import the EDD's HTML mapping table into a document when you import the element definitions into template or other document. See "Specifying whether to transfer HTML mapping tables" on page 89.
- Element: Begins the definition of an element. See "Writing element definitions" on page 91.
- FormatChangeList: Begins the definition of a named format change list. You can use one of these lists to describe a set of changes to format properties and then refer to the list from element definitions. This is helpful when two or more elements use the same set of changes because you need to describe the changes only once. See "Defining a format change list" on page 163.
- Para: Inserts a paragraph so that you can comment the EDD. This is useful for adding descriptions of groups of elements or sections of the EDD. See "Organizing and commenting an EDD" on page 90.
- Section: Creates a section so that you can group element definitions or format change lists in the EDD. By using sections, you can divide a large EDD into more manageable parts. A Head child element is inserted automatically with Section. See "Organizing and commenting an EDD" on page 90.
- FormatChangeListLimits: Begins a specification of minimum and maximum limits on relative values used in format change lists and text format rules in the EDD. See "Setting minimum and maximum limits on properties" on page 165.

All elements in the catalog

This section lists all of the elements in an EDD's Element Catalog—except for a few very low-level elements for completing a formatting specification, such as Left and Right (for Alignment) and Yes and No (for ChangeBars). Click an element tag to go to the discussion of the element in one of the EDD chapters.

Element tag	Purpose of the element
Alignment	Side head alignment
AlignOn	Character for a decimal tab stop
AllContextsRule	Context specification in a format rule
AnchoredFrame	Initial content type for a graphic element
Angle	Font angle
Attribute	Attribute definition
AttributeList	List of attribute definitions
AttributeValue	Attribute value in a prefix or suffix rule
AutoInsertions	Structure rule for automatically nested descendants
AutonumberFormat	Autonumber format
AutonumCharFormat	Character format for an autonumber
Bottom	Bottom cell margin for a table
BottomCellMarginLimits	Limits on all bottom cell margins
Case	Capitalization style
CellMarginLimits	Limits on all cell margins
CellMargins	Set of table cell margins
ChangeBars	Change bars in page margins
CharacterFormatTag	Reference to a character format in a format rule
Choice	Attribute type
Choices	List of values for a Choice attribute
Color	Text color
Comments	Explanatory paragraph in an element definition
Container	Element type
ContextLabel	Label for a formatting variation of an element
ContextRule	Context specification in a format rule
CountAncestors	Ancestor to count in a level rule
CreateFormats	Create formats when importing an EDD
CrossReference	Element type
CrossReferenceFormat	Initial cross-reference format
Default	Default value for an attribute
DefaultPunctuation	Default punctuation for a side or run-in head
DefaultSystemVariable	Default variable for a system variable element
DoNotCreateFormats	Do not create formats when importing an EDD
Element	Element definition

Element tag	Purpose of the element
ElementCatalog	Highest-level element in an EDD
ElementPgfFormatTag	Reference to a base paragraph format
Else	Clause in a context or level rule
ElseIf	Clause in a context or level rule
Equation	Element type
Exclusion	Structure rule for excluding an element from a defined element or its descendants
Family	Font family
FirstIndent	First-line paragraph indent
FirstIndentChange	First-line paragraph indent, as a change to the current indent
FirstIndentLimits	Limits on all first-line paragraph indents
FirstIndentRelative	First-line paragraph indent, relative to a left indent
FirstParagraphRules	Text format rules for first paragraph in an element
FontSizeLimits	Limits on all font sizes
Footnote	Element type
FormatChangeList	List of changes to text formatting properties
FormatChangeListLimits	List of limits on values in format change lists
FormatChangeListTag	Reference to a format change list in a format rule
FrameAbove	Reference to a graphic frame above a paragraph
FrameBelow	Reference to a graphic frame below a paragraph
FramePosition	Position for a graphic frame
GeneralRule	Structure rule for allowed content in an element
Graphic	Element type
Head	Head for a section in an EDD
Height	Line spacing
HeightChange	Line spacing, as a change to the current height
Hyphenate	Automatic hyphenation
Hyphenation	Set of hyphenation properties
IDReference	Attribute type
IDReferences	Attribute type
If	Clause in a context or level rule
ImportedGraphicFile	Initial content type for a graphic element
Inclusion	Structure rule for allowing an element in a defined element or its descendants

Element tag	Purpose of the element
Indents	Set of paragraph indents
InitialObjectFormat	Object format rule for a graphic, marker, cross-reference, or equation
InitialStructurePattern	Structure rule for automatic tagging in a table
InitialTableFormat	Object format rule for a table
InsertChild	Child element for automatic insertion
InsertNestedChild	Nested child element for automatic insertion
Integer	Attribute type
Integers	Attribute type
KeepWithNext	Keep paragraph with next paragraph
KeepWithPrevious	Keep paragraph with previous paragraph
Language	Language for hyphenation and spell-checking
Large	Initial equation size
LastParagraphRules	Text format rules for last paragraph in an element
Leader	Tab leader character
Left	Left cell margin for a table
LeftCellMarginLimits	Limits on all left cell margins
LeftIndent	Left paragraph indent
LeftIndentChange	Left paragraph indent, as a change to the current indent
LeftIndentLimits	Limits on all left paragraph indents
LetterSpacing	Additional letter spacing to optimize word spacing
LevelRule	Level specification in a format rule
LineSpacing	Set of line spacing properties
LineSpacingLimits	Limits on all line spacing
Marker	Element type
MarkerType	Initial marker type
MaxAdjacent	Maximum adjacent hyphenated lines
Maximum	Specification for a limit on a value
Maximum	Maximum word spacing
Medium	Initial equation size
Minimum	Specification for a limit on a value
Minimum	Minimum word spacing
MoveAllTabStopsBy	Relative change position for all tab stops
Name	Attribute name

Element tag	Purpose of the element
NoAdditionalFormatting	No text formatting changes
NoAutonumber	No autonumbering
OffsetHorizontal	Horizontal text range offset
OffsetVertical	Vertical text range offset
Optimum	Optimum word spacing
Optional	Optional attribute value
Outline	Outline text style
Overline	Overline text style
PairKerning	Pair kerning
Para	Explanatory paragraph in an EDD, outside an element definition
ParagraphFormatTag	Reference to a paragraph format in a format rule
ParagraphFormatting	Formatting an element as a paragraph
ParagraphSpacing	Set of paragraph spacing properties
PgfAlignment	Paragraph alignment
Placement	Paragraph placement on a page
Position	Autonumber position in a paragraph
Prefix	Text string for a prefix
PrefixRules	Rules for a prefix
PropertiesAdvanced	Set of formatting properties for frames, hyphenation, and word spacing
PropertiesBasic	Set of formatting properties for indents, alignment, tab stops, and line and paragraph spacing
PropertiesFont	Set of formatting properties for font and text style
PropertiesNumbering	Set of formatting properties for autonumbering
PropertiesPagination	Set of formatting properties for paragraph placement
PropertiesTableCell	Set of formatting properties for table cells
Range	Range of values for a numeric attribute
ReadOnly	Read-only attribute
Real	Attribute type
Reals	Attribute type
RelativeTabStopPosition	Tab stop position, relative to a left indent
Required	Required attribute value
Right	Right cell margin for a table

Element tag	Purpose of the element
RightCellMarginLimits	Limits on all right cell margins
RightIndent	Right paragraph indent
RightIndentChange	Right paragraph indent, as a change to the current value
RightIndentLimits	Limits on all right paragraph indents
Rubi	Element type
Rubi Group	Element type
Section	Section in an EDD
Shadow	Text shadowing
ShortestPrefix	Shortest prefix in a hyphenated word
ShortestSuffix	Shortest suffix in a hyphenated word
ShortestWord	Shortest hyphenated word
Size	Text size
SizeChange	Text size, as a change to the current size
Small	Initial equation size
SpaceAbove	Space above a paragraph
SpaceAboveChange	Space above a paragraph, as a change to the current spacing
SpaceAboveLimits	Limits on all space above paragraphs
SpaceBelow	Space below a paragraph
SpaceBelowChange	Space below a paragraph, as a change to the current spacing
SpaceBelowLimits	Limits on all space below paragraphs
Specification	Child element for If, Else, or ElseIf in a context or level rule
Spread	Text spread
SpreadChange	Text spread, as a change to the current spread
StartPosition	Paragraph start position in a column
StopCountingAt	Ancestor to stop counting at in a level rule
Strikethrough	Strikethrough text style
StructuredApplication	Reference to an SGML application for an EDD
Subrule	Nested format rule
Suffix	Text string for a suffix
SuffixRules	Rules for a suffix
Superscript Subscript	Superscript or subscript text style

Element tag	Purpose of the element
SystemVariable	Element type
SystemVariableFormatRule	Object format rule for a system variable
TabAlignment	Tab stop alignment
Table	Element type
TableBody	Element type
TableCell	Element type
TableFooting	Element type
TableFormat	Initial table format
TableHeading	Element type
TableRow	Element type
TableTitle	Element type
TabStop	Tab stop definition
TabStopPosition	Tab stop position
TabStopPositionLimits	Limits on all tab stop positions
TabStops	Set of tab stop definitions
Tag	Element tag
TextFormatRules	Text format rules
TextRangeFormatting	Formatting an element as a text range
Top	Top cell margin for a table
TopCellMarginLImits	Limits on all top cell margins
Tracking	The space between characters
TrackingChange	The space added to the current tracking
Underline	Underline text style
UseSystemVariable	Variable for a system variable element
ValidHighestLevel	Validity at the highest level in a flow
Variation	Font variation
VerticalAlignment	Text alignment in a table cell
Weight	Font weight
WidowOrphanLines	Minimum lines from a paragraph that appear alone in a column
WordSpacing	Set of word spacing properties

Defining preliminary settings in an EDD

The beginning of an EDD can have three settings that define general characteristics for the EDD: Version, CreateFormats (or DoNotCreateFormats), and

StructuredApplication. FrameMaker supplies the version number, which is not editable. The other settings you can define or change yourself.

Specifying whether to create formats automatically

When you import element definitions from an EDD into a template or other document, your definitions may refer to paragraph formats, character formats, table formats, or cross-reference formats that are not already in the template. You can have FrameMaker create any missing, named formats in the template when you import the definitions. (The new formats will have default properties. In many cases, you or the template designer will probably need to change the properties.)

A new EDD has a CreateFormats element, which specifies that formats will be created automatically on import. If you do not want FrameMaker to create the formats, select the CreateFormats element and change it to DoNotCreateFormats. Whichever element you use must come before any element definitions, format change lists, sections, and paragraphs.

Specifying whether to transfer HTML mapping tables

FrameMaker products can save documents as HTML. To do this, each document can have a mapping table on its HTML reference page. An EDD can also have an HTML mapping table on its EDD_HTML reference page. When importing an element definition into a document, you can have FrameMaker also import the EDD mapping table. If you are importing into a book, FrameMaker imports the mapping table to the BookHTML reference page of the first component file in the book.

An EDD can include an ImportHTMLMapping element, which tells the software to import the mapping table from the EDD to the HTML reference page of any document that imports the EDD. If you do not want documents to import the EDD's mapping table, select the ImportHTMLMapping element and change it to the DoNotImportHTMLMapping element.

When exporting a document's element catalog as an EDD, if the document has a mapping table on the HTML reference page, it will export that table to the EDD, and the EDD will contain an ImportHTMLMapping element.

For information on HTML mapping tables, see *FrameMaker User Guide*.

Setting a structure application

If you are working with XML or SGML, you need to specify which structure application to associate with the EDD. A structure application defines information such as a DTD, an SGML declaration, a read/write rules document, an application definition file, entity catalogs, and a FrameMaker template (which has the elements from your EDD). FrameMaker uses

the application when you translate between a DTD and an EDD and when an end user shares documents between XML or SGML and FrameMaker.

> **Important:** The DTDs for SGML and XML are significantly different. For this reason you should always use XML structure applications for XML files, and SGML structure applications for SGML files.

When you first convert a DTD to an EDD using File>Structure Tools>Open DTD, you can select a structure application for the EDD. The new EDD has a `StructuredApplication` element with the name of the application. If you select <No Application> when you convert the DTD, the new EDD does not have this element.

To set a structure application in an EDD that does not have an application, insert a `StructuredApplication` element and type the name of the application. This element must come before any sections, paragraphs, element definitions, and format change lists. To change an application already set in an EDD, edit the name in the `StructureApplication` element.

All documents that use the EDD are also associated with the structure application. Users can change to a different application in an individual document by using the Set Structure Application command (File menu).

For information on the parts of a structure application and the process of developing one, see Chapter 6, “Creating a Structure Application.”

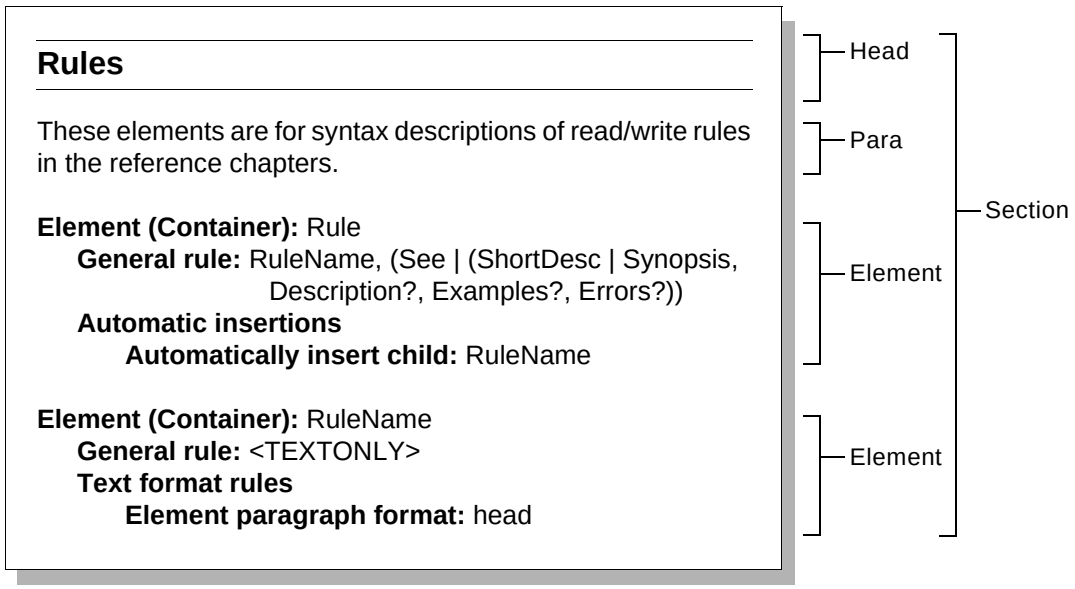
Organizing and commenting an EDD

You can add optional `Section` elements to an EDD to group element definitions and format change lists, and you can add optional `Para` elements to provide additional information about the EDD. Using a few `Section` and `Para` elements judiciously can make your EDD more readable and easier to maintain.

Sections are particularly useful when an EDD has many definitions. You may want to group and label the definitions by element type, with format change lists in another section. For example, your sections might be named `Containers`, `Tables`, `Objects`, and `Format Change Lists`.

Add explanatory paragraphs wherever needed to provide descriptive information about a group of elements. This information can be helpful to someone maintaining the EDD later. (You can also add comments to individual element definitions. See “Writing element definitions,” next.)

This example shows a part of an EDD that is organized in a section and has an introductory paragraph for the section:



To add a section, insert a `Section` element in `ElementCatalog` or in another `Section`. A `Head` child element is inserted automatically. Type a label for the section in the `Head` element. You can insert `Element`, `FormatChangeList`, `Para`, and other `Section` elements as child elements of a parent `Section` element. If necessary, wrap or move existing elements into the section.

To add a paragraph, insert a `Para` element in `ElementCatalog` or in a `Section`, and type the paragraph.

`Section` and `Para` elements are ignored when you import element definitions into a template or other document. When you save an EDD as a DTD, they are written as comments.

Writing element definitions

To write an element definition, begin by inserting an element called `Element` in `ElementCatalog` or in a `Section`. Then use the Element Catalog or the status bar as a guide as you type text and insert child elements to create a valid definition.

Elements in FrameMaker fall into two basic groups: containers, tables, and footnotes; and object elements such as markers and equations. Containers, tables, and footnotes can hold text and other elements, whereas an object element holds one of its specified type of object and nothing more. These differences are reflected in the way you write the element definitions:

- Containers, tables, and footnotes must have content rules that define valid contents for the element. Object elements do not have content rules.

- Containers, tables, and footnotes can have text format rules that specify font and paragraph formatting for text in the element and its descendants. Object elements can have an object format rule that specifies a single property, such as a marker type or an equation size. (A table can have both text format rules and an object format rule.)

In other respects, element definitions are alike for the two groups of elements. They must all have a unique element tag and a declared type, and they can have attribute definitions and comments.

About element tags

When naming an element, give the element a tag that is self-explanatory and unique. A user will need to recognize the purpose of the element to select it in the Element Catalog and use it properly. Element tags are case-sensitive, and they can contain white space but none of these special characters:

```
( ) & | , * + ? < > % [ ] = ! ; : { } "
```

An element tag can have up to 255 characters in FrameMaker, but you should try to keep the tags concise. The default width of the Element Catalog a user sees shows the first 14 characters of a tag. (If you are using context labels, the maximum length is 255 for the tag and label together.)

The Element Catalog in a document shows the currently available elements in alphabetical order (unless the end user is displaying a customized list). In some cases, especially if the list of elements is long, you may want to name elements in a way that will group them logically in the catalog. For example, if you have two types of table elements you might name them `TblSamples` and `TblStandard` to display them together in the catalog. If you do begin any tags the same way, keep the first part of the tag as short as possible.

Don't begin tags the same way unless you need to for grouping. A user can usually find elements in the catalog if the tags are distinct, and the user may want to type in a unique beginning string to identify a tag for a quick key command (such as Control-1 for Insert Element).

SGML: If you plan to export documents to SGML, define element tags that conform to the naming rules and the maximum name length permitted by the concrete syntax you'll be using in SGML. If you prefer tags that do not adhere to the SGML conventions, you can provide read/write rules to convert them to SGML equivalents when you export. For more information on element names in SGML, see "Naming elements and attributes" on page 231.

Guidelines for writing element definitions

Here are a few points to keep in mind when writing element definitions:

- In most cases, you should work iteratively in the EDD. Write at least some of the definitions, import the definitions into a template, test the template on sample documents, and repeat the process as necessary.

For information on importing and testing the definitions, see "Creating an Element Catalog in a template" on page 103.

- In many EDDs, the most complicated part of the definitions is the format rules (particularly text format rules). The first time you work in a particular EDD, consider defining just the structure rules and attribute definitions and testing only the structure and attributes at that point. Then you can go back and add the format rules and test them in a separate pass.
- Provide a highest-level element for each structured flow possible in the documents. For a book file, provide a highest-level element for the book and for each possible book component (such as Front, Chapter, and Index).
- After writing element definitions, validate the EDD before importing the definitions so that you find missing elements and content errors.
- Remember that the formats you refer to in element definitions must be stored in the template. If you are working with a template designer, you need to coordinate the tags and properties of the formats with the designer.
- Create user variables for text in the EDD that you use again and again. For example, if several elements have the same general rule, define a variable for the general rule. Then, if necessary, you can change the general rules for all the elements by redefining the variable definition.

See also "Keyboard shortcuts for working in an EDD" on page 102.

Defining a container, table or footnote element

Containers are general-purpose elements that you define for text, child elements, or a combination of the two. Paragraphs, text ranges, heads, sections, and chapters are common examples of containers. In a typical document, most elements are containers.

Tables, table parts (titles, headings, bodies, footings, rows, and cells), and footnotes are similar to containers in that they can hold child elements and in some cases text. But these elements are for a specific purpose—for a table or a footnote—and can be used only for that purpose in a document.

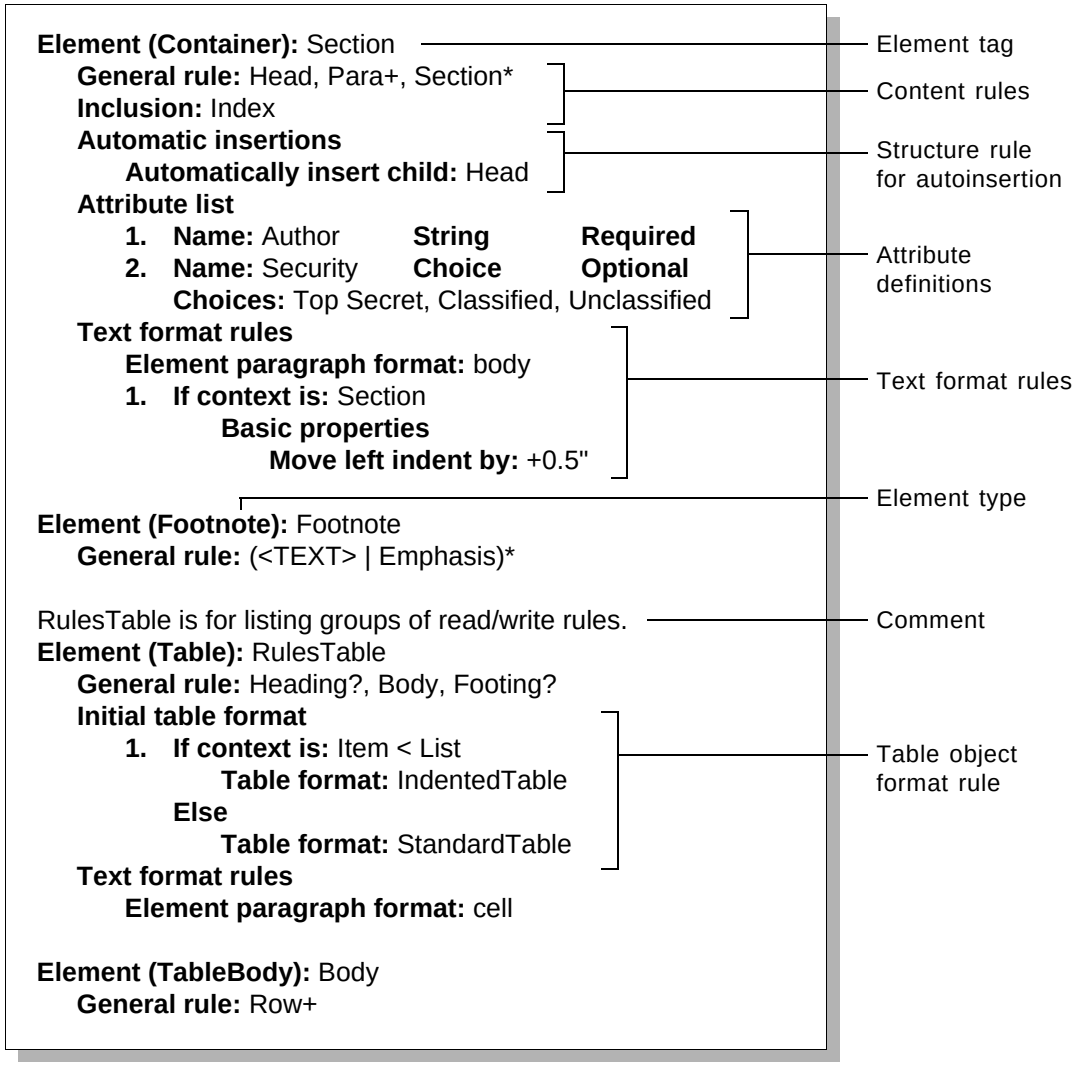
An element definition for a container, table, table part, or footnote specifies a unique element tag and element type and can also have any of these items:

- A comment that describes the element
- Content rules that describe valid contents for the element or its descendants (the general rule part of this is required). For each structured flow in the documents, at least one container needs a rule specifying that the element is valid at the highest level
- For a container, additional structure rules that provide initial contents for new instances of the element. For a table, a tagging pattern that specifies the element tags assigned to the rows and cells an end user creates with a new table

- Attribute definitions that specify attributes to store descriptive information with the element
- Text format rules that determine how to format text in the element or its descendants
- For a table, an object format rule that determines an initial table format for new instances of the element

Examples

These are definitions for containers, tables, table parts, and footnotes:



Basic steps

This section gives an overview of the steps for defining a container, table, table part, or footnote. Refer to the chapters that follow for detailed descriptions of the syntax and more examples.

Note that the steps in this section suggest an order for the rules in a definition, but in some cases you can write the rules in a different order. (For example, the element for highest-level validity can go before or after the general rule.) Use the Element Catalog as a guide for inserting elements in a valid order.

1. **Insert an `Element` element in the highest-level element of the EDD (`ElementCatalog`) or in a `Section`. Then type a tag in the `Tag` element.**

   When you insert `Element`, the `Tag` child element is inserted automatically. For guidelines on providing a tag, see "About element tags" on page 92.

2. **(Optional) If you want to include a comment for the definition, insert a `Comments` element before `Tag` and type the comment.**

   The comment appears just above the definition's tag line. If you include a comment, place the insertion point right after `Tag` when you're finished.

3. **Insert an element to specify the element type.**

   An element type determines what other child elements will be available as you write the definition.

For this type	Insert the element
General-purpose element for holding text, child elements, or both	`Container`
Table	`Table`
Element for holding rows in a structured table	`TableHeading`, `TableBody`, or `TableFooting`
Element for holding cells in a structured table	`TableRow`
Element for holding text in a structured table	`TableTitle` or `TableCell`
Footnote	`Footnote`

   When you insert one of these elements, the name of the type appears in parentheses before the tag and a `GeneralRule` child element is inserted automatically.

4. **Type the general rule in the `GeneralRule` element to define allowed contents for the element.**

   A general rule describes the child elements the element can contain, whether the child elements are required or optional, and the order in which the child elements can occur. It also specifies whether the element can have text.

If you do not specify a general rule, FrameMaker gives the element a default general rule that depends on the element's type. To use a default rule, leave the `GeneralRule` element empty (but do not delete `GeneralRule` or the definition will be invalid). These are the default general rules:

Element type	Default general rule
Container	`<ANY>`
Table	`TITLE?, HEADING?, BODY, FOOTING?`
Table heading, body, or footing	`ROW+`
Table row	`CELL+`
Footnote, table title, or table cell	`<TEXT>`

For information on the syntax and restrictions of general rules, see "Writing an EDD general rule" on page 111.

5. (Optional) Define other content rules as necessary.

Every structured flow in a document needs one highest-level container element. If the element you're defining is a container, you can insert a `ValidHighestLevel` child element to allow the element to be at the highest level. For more information, see "Specifying validity at the highest level in a flow" on page 116.

SGML, only: For a container, table, table part, or footnote, you can define inclusions and exclusions. An inclusion is an element that can occur anywhere inside the defined element or its descendants, and an exclusion is an element that cannot occur anywhere in the element or its descendants. For each element you want to include or exclude, insert an `Inclusion` or `Exclusion` element and type the element tag. For more information, see "Adding inclusions and exclusions" on page 116.

6. (Optional) Write additional structure rules to specify initial contents or tagging for new instances of the element.

For a container, you can define nested descendants that will appear automatically with the element in a document. Insert an `AutoInsertions` element, and for the first child insert an `InsertChild` element and type the element tag. Then for each nested descendant, insert an `InsertNestedChild` element and type the tag. For more information, see "Inserting descendants automatically in containers" on page 119.

For a table, heading, body, footing, or row, you can define element tags that will be used in row or cell elements in the table or table part. Insert an `InitialStructurePattern` element, and then type the tags of the child elements, separated by commas. For more information, see "Inserting table parts automatically in tables" on page 120.

7. (Optional) Write attribute definitions to define attributes that can store additional information about instances of the element.

In FrameMaker, attributes can be used to record information such as the current status of an element, to maintain IDs and ID references for cross-referencing between elements, and to allow an element to be formatted using the current value of its attribute.

Insert an `AttributeList` element. The first `Attribute` child element is inserted automatically. Define the first attribute, and then insert and define additional `Attribute` elements as necessary. For more information, see Chapter 11, "Attribute Definitions."

8. (Optional) Write text format rules to describe how to format text in the element or its descendants.

Text format rules can refer to a paragraph format to use as a "base" format for the element and can specify context-dependent changes to the format in use. If you write text format rules for a table, heading, body, footing, or row, the rules specify formatting only for text in descendant titles and cells.

Insert a `TextFormatRules` element. Then to specify a paragraph format, insert an `ElementPgfFormatTag` element as the first child element of `TextFormatRules` and type the format tag. For each set of formatting changes, insert a context element (`AllContextsRule`, `ContextRule`, or `LevelRule`), specify the context, and define the changes for the context.

For containers, you can also write text format rules for the first and last paragraphs in the element, and you can define and format a prefix or suffix to appear at the beginning or end of the element. Insert `FirstParagraphRules`, `LastParagraphRules`, `PrefixRules`, or `SuffixRules`, and define the context and formatting specifications.

For information on the syntax of text format rules and how rules can be inherited from ancestors, see Chapter 10, "Text Format Rules for Containers, Tables, and Footnotes,"

9. (Optional) If the element is a table, write an object format rule to define an initial table format for new instances of the table.

A table format determines the basic appearance of the table—such as indentation and alignment, margins and shading in cells, and ruling between columns and rows.

Insert an `InitialTableFormat` element, and insert and define context elements as necessary. Type the tag of a table format for each context. For more information, see "Setting a table format" on page 190.

Defining a Rubi group element

Documents that include Japanese text most likely require Rubi to express the pronunciation of certain words. A *Rubi group* is an element that contains such text. The Rubi group includes a Rubi Group element for the base word (Oyamoji), and a Rubi element for the phoenetic spelling (Rubi) to the Oyamoji. These elements can be used only for a Rubi group.

The Rubi element must be the last child element of the Rubi Group element. The Rubi Group and Rubi elements can contain other child elements. For example, the Oyamoji can be in a container of its own, or you can insert footnotes within Oyamoji or Rubi text. Note that the JIS rules for a Rubi group state that the group cannot extend across a line break.

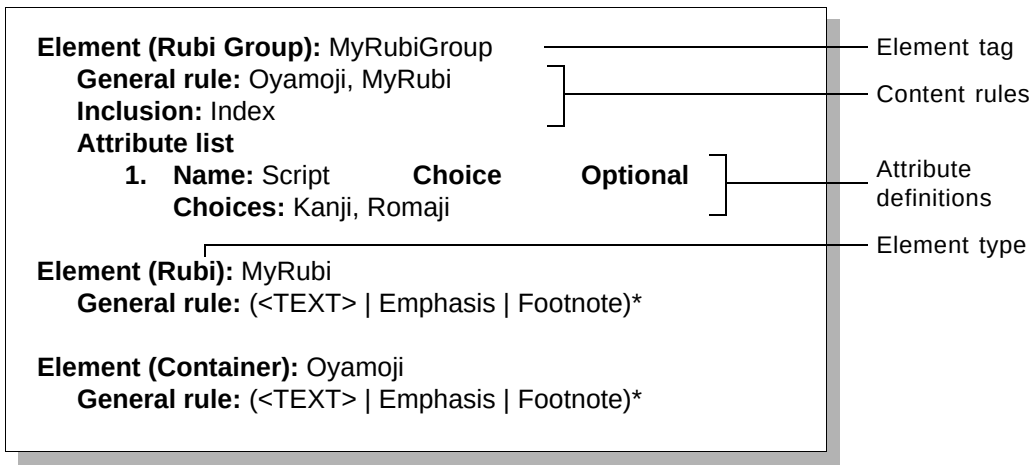
Therefore, the children of either the Rubi Group or the Rubi elements should be text range elements; FrameMaker will not insert a line break or a paragraph break within a Rubi Group.

An element definition for a Rubi group specifies a unique element tag and an element type for both the Rubi group and the Rubi text, and they can also have any of these items:

- A comment that describes the element
- Content rules that describe valid contents for the element or its descendants (the general rule part of this is required). A Rubi Group element cannot be valid at the highest level.
- For a Rubi group, an initial structure pattern that specifies the element tag assigned to the child Rubi element
- Attribute definitions that specify attributes to store descriptive information with the element
- Text format rules that determine how to format text in the element or its descendants

Examples

These are definitions for a Rubi group, a Rubi element, and a container for Oyamoji text:



Basic steps

The steps for creating a Rubi Group element and a Rubi element are very much the same as the steps for defining a container, table, table part, or footnote element. Use the Element Catalog as a guide for inserting elements in a valid order. Refer to the chapters that follow for detailed descriptions of the syntax and more examples.

Note that you cannot insert an `AutoInsertions` element to specify auto insertions for a Rubi group, but you can insert an `InitialStructurePattern` element, and then type the tag of the child Rubi element. For more information, see "Inserting Rubi elements automatically in Rubi groups" on page 123.

Defining an object element

A FrameMaker document uses special object elements for tables, graphics, markers, cross-references, equations, system variables, footnotes, Rubi groups, and Rubi text. An instance of the element holds exactly one of the specified objects.

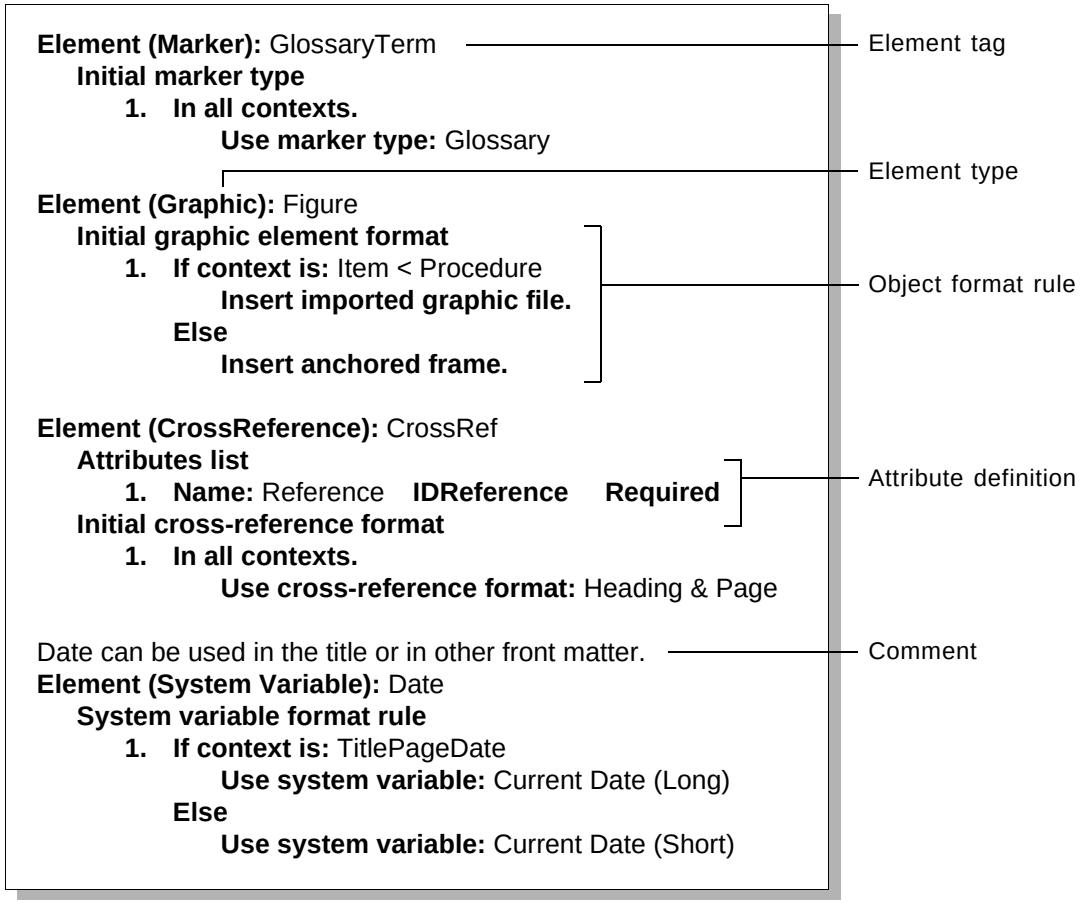
This section explains how to define object elements except for tables. Because tables can hold other elements, they are similar to containers and are described in "Defining a container, table or footnote element" on page 93.

An element definition for a FrameMaker object specifies a unique element tag and element type and can also have any of these items:

- A comment that describes the element
- Attribute definitions that specify attributes to store descriptive information with the element
- An object format rule that determines a formatting property, such as a marker type or equation size, for new instances of the element

Examples

These are definitions for object elements:



Basic steps

This section gives an overview of the steps for defining an object element. Refer to the chapters that follow for detailed descriptions of the syntax and more examples.

1. **Insert an `Element` element in the highest-level element of the EDD (`ElementCatalog`) or in a `Section`. Then type a tag in the `Tag` element.**

   When you insert `Element`, the `Tag` child element is inserted automatically. For guidelines on providing a tag, see "About element tags" on page 92.

2. **(Optional) If you want to include a comment for the definition, insert a `Comments` element before `Tag` and type the comment.**

   The comment appears just above the definition's tag line. If you include a comment, place the insertion point right after `Tag` when you're finished.

3. **Insert an element to specify the element type.**

An element type determines what other child elements will be available as you write the definition.

For this object	Insert the element
Anchored frame	`Graphic`
Cross-reference	`CrossReference`
Equation	`Equation`
Imported graphic file	`Graphic`
Marker	`Marker`
System variable	`SystemVariable`

When you insert one of these elements, the name of the type appears in parentheses before the tag.

4. **(Optional) Write attribute definitions to define attributes that can store additional information about instances of the element.**

   In FrameMaker, attributes can be used to record information such as the current status of an element, to maintain IDs and ID references for cross-referencing between elements, and to allow an element to be formatted using the current value of its attribute.

   Insert an `AttributeList` element. The first `Attribute` child element is inserted automatically. Define the first attribute, and then insert and define additional Attribute elements as necessary. For more information, see Chapter 11, “Attribute Definitions.”

5. **(Optional) Write an object format rule to define a formatting property for new instances of the element.**

   These are the properties you can specify in an object format rule:

Element type	Property
Graphic	Anchored frame or imported graphic file
Cross-reference	Cross-reference format
Equation	Equation size
Marker	Marker type
System variable	System variable name

   With the exception of system variable names, these properties are not binding. An end user can change a marker type, cross-reference format, or other property at any time, and the change is not considered to be a format rule override. (The user can remove all format rule overrides when he or she reimports element definitions.) A user cannot change the variable name for a system variable element.

   Unlike text format rules, an object format rule defines the property only for the current element. Because object elements do not have descendants, the object rule is not passed on through a hierarchy to other elements.

Insert an `InitialObjectFormat` or `SystemVariableFormatRule` element, and insert and define context elements as necessary. Define a formatting property for each context. For more information, see Chapter 12, "Object Format Rules."

Keyboard shortcuts for working in an EDD

This section gives some of the most useful keyboard shortcuts for working in an EDD (or in any other structured document). For a complete list of the shortcuts available, see the FrameMaker quick reference.

Editing structure

Use these shortcuts for editing the structure of an EDD—for example, inserting, wrapping, changing, and rearranging elements (Note: for these shortcuts to function, an element must already be selected).

To	Press
Insert an element at the current location	Control-1 (one) or Esc Ei
Wrap an element around the current selection	Control-2 or Esc Ew
Change the current element	Control-3 or Esc Ec
Unwrap the current element	Esc Eu
Merge into the first element (when more than one element is selected)	Esc Em
Merge into the last element (when more than one element is selected)	Esc EM
Split the current element	Esc Es
Repeat the last insert, wrap, or change element command	Esc ee
Move the element to the next higher level (promote)	Esc EP
Move the element to the next lower level (demote)	Esc ED
Collapse or expand the current element in the Structure View	Esc Ex
Collapse or expand the element's siblings in the Structure View	Esc EX
Transpose the current element with the previous one	Esc ET
Transpose the current element with the next one	Esc Et

Moving around the structure

Use these shortcuts to move the insertion point around the structure of an EDD.

To move the insertion point	Press
To the next element down in the structure	Esc sD or Meta-down arrow
To the previous element up in the structure	Esc sU or Meta-up arrow
To the beginning of the next element's contents	Esc sN or Meta-right arrow
To the start of the current element	Esc sS
To the end of the current element	Esc sE
Just before the current element's parent	Esc sB

Creating an Element Catalog in a template

A FrameMaker template stores all of the catalogs and properties that end users need for their documents. This can include an Element Catalog built from your EDD—as well as a Paragraph Catalog, Character Catalog, Table Catalog, and a variety of other properties that affect the appearance of documents, such as page layouts and cross-reference formats. You create a template's Element Catalog by importing the definitions from the EDD into the template. (In many cases, the other catalogs and properties are set and maintained by a template designer.)

When end users need to work with the elements you've defined, they can create new documents from the template or import the Element Catalog from the template into their existing documents. Users do not normally need to see and interpret definitions in the EDD.

You can also make the template part of a structure application. To do this, insert a `Template` element in the application definition in the `structapps.fm` file and type the pathname of the template. When end users open a markup document that uses that application, FrameMaker applies structure and formatting from the template to the document. For more information, see "Application definition file" on page 50.

Many of the formats stored in a template may be used by format rules in your element definitions; for example, text format rules can refer to paragraph formats, and table object format rules can refer to table formats. If you are working with a template designer, you'll need to coordinate the tags and properties of these formats with the designer. If you are acting as template designer as well as EDD developer, you'll need to define the formats yourself. For advice on planning and designing the parts of a template, see the *FrameMaker User Guide*.

Even after importing element definitions into a template, you will probably want to keep the EDD as a separate file for maintenance purposes.

Importing element definitions

To import element definitions, choose Element Definitions from the File>Import submenu in the template. Select the EDD in the Import from Document dialog box. (The EDD must be open to appear in this dialog box.) If the EDD is invalid, an alert tells you to correct the validation errors and then import the EDD.

If you import definitions into a template that does not yet have an Element Catalog, FrameMaker creates a new catalog for the template. If you import definitions into a template that already has a catalog, FrameMaker completely replaces the old catalog with your new one.

Log files for imported element definitions

If FrameMaker encounters any problems while importing element definitions, it produces a log file of warnings and errors. A warning is a notification of a potential problem, but it does not necessarily mean something is wrong with the EDD. An error indicates an actual problem in the processing; some errors can cause the processing to stop altogether.

Some warning messages describe formats that FrameMaker needs to create because they are referenced in the EDD but not yet stored in the template. If you do not want FrameMaker to create formats automatically when you import element definitions, you can insert a `DoNotCreateFormats` element as a child element of `ElementCatalog` in the EDD. (If the EDD already has a `CreateFormats` element, select it and change it to `DoNotCreateFormats`.)

A log file can also have messages for:

- defined elements not used in any content rules
- elements referenced in content rules that are not defined
- definitions that have syntax errors

This is an example of a message in a log file:

```
Referenced element "Part" is undefined.
```

All messages in the log file have a hypertext link to the EDD. You can click a message to go to the line with the problem in the EDD.

A log file is initially locked so that you can click in it to use the hypertext links. If you want to save the log file, you must first unlock it by pressing Esc Flk. (Press Esc Flk again to relock the file.) For general information on FrameMaker log files, see “Log files” on page 71.

Debugging element definitions

You should test element definitions thoroughly before delivering a structured template to end users. First correct any errors in the EDD that are identified in the log file and reimport the definitions until FrameMaker no longer finds errors. Then import the Element Catalog from the template into a sample document that contains representative text and check to

see that the elements behave in the way you expect. You may need to test the elements in the document, make revisions in the EDD, reimport the elements into the EDD, reimport the catalog into the document, and repeat the process as necessary.

Here are a few tasks to go through while testing elements in a sample document:

- Insert and wrap a variety of elements in the document. If you've defined any containers to have child elements added automatically, make sure the child elements also appear when you insert the containers.
- Move elements around in the document. Check the Structure View to see that the elements are valid only where they should be.
- Enter attribute values in elements that allow them. Check the Structure View to make sure that the types of values you want to add are valid according to the attribute's definition.
- Once you have the format rules in the EDD, move elements around to verify that they are formatted correctly according to context. Check to see that extra formatting items such as prefixes and autonumbers appear where they should.

For help on identifying syntax and context errors in an element definition, see:

- "Debugging structure rules" on page 123
- "Debugging text format rules" on page 166
- "Debugging object format rules" on page 197

Saving an EDD as a DTD for export

If your end users will be saving FrameMaker documents as markup and you did not begin the development process with a DTD, you need to save your EDD as a DTD so that users will have a DTD for exported documents. FrameMaker creates a new DTD with element declarations and attribute definition list declarations that correspond to element and attribute definitions in the EDD.

You need to define a finished DTD as part of a structure application before your end users can save documents as markup. To do this, insert a `DTD` element in the application definition in `structapps.fm` and type the pathname of the DTD. For more information, see "Application definition file" on page 50.

Read/write rules and the new DTD

When creating a DTD from an EDD, we recommend that you first create an initial DTD with no read/write rules—or with only a subset of the rules if you have some already developed. This lets you see how FrameMaker translates the EDD with little or no help from your rules.

Once you have both an EDD and a DTD, you can refine the translation in an iterative process of developing read/write rules. First analyze the EDD and new DTD together to plan how to modify the translation with rules. Then develop at least some of your rules, update

the DTD from the EDD using the rules, test the results on sample documents, and repeat the process as many times as necessary. You may find it easiest to write and test only a few rules during each iteration. For a more detailed discussion of this process, see "Task 3. Creating read/write rules" on page 35.

You develop read/write rules in a special rules document that is part of a structure application. When you create a DTD from an EDD, you can specify which application (and hence which set of rules) to use with the DTD. For information on developing a read/write rules document, see Chapter 14, "Read/Write Rules and Their Syntax."

Creating a DTD from an EDD

To create a DTD from an EDD, choose Save as DTD from the File>Developer Tools submenu in the EDD. Specify a location in the Save as DTD dialog box.

If the Use Application dialog box appears, select a structure application for the DTD. (Use Application appears only if no application is specified in the EDD.) If you're creating an initial DTD without read/write rules, select <No Application> for a default structure application with no rules. To specify structure application for the DTD later, you can use Save as DTD again and this time select the application.

What happens during translation

An EDD has element and attribute definitions that correspond to element and attribute definition list declarations in a DTD. When you translate an EDD to a DTD, FrameMaker makes assumptions about how the constructs from the EDD should be represented. FrameMaker reads through the entire EDD, processing elements and their attributes one at a time. In the absence of read/write rules, the software translates a FrameMaker element definition to an XML or SGML element declaration of the same name, and it produces an attribute list declaration for each attribute defined for the element.

FrameMaker writes other EDD constructs in various ways; for example, variables become entities and markers become processing instructions. Comments and `Section` and `Para` elements in the EDD become comments.

Note that EDDs include more semantic information about the usage of elements than DTDs do. For example, an EDD may have special element types corresponding to markers, system variables, and graphics. The declarations in a DTD created by FrameMaker reflect this information.

For details on the translation of each type of element, see "Translating between markup data and FrameMaker" on page 199.

> **Important:** When exporting an EDD as a DTD, if the new DTD file has the same name as the old DTD file, FrameMaker can save a backup version of the old DTD. To use this feature, you must turn on Automatic Backup on Save in the Preferences dialog box. On the Macintosh, this option is turned off by default.

SGML declarations

FrameMaker runs a new DTD through an XML or SGML parser. In the process, it may identify errors in the syntax of the DTD. Two of the most common errors are invalid SGML names and an inappropriate SGML declaration. You can use read/write rules to translate FrameMaker element tags to valid SGML names.

For SGML, if the default SGML declaration that FrameMaker provides is not appropriate for your DTD, you can modify the declaration to avoid capacity and quantity errors. The default SGML declaration for FrameMaker uses the reference *concrete syntax* and the reference quantity set. To change to a different declaration, insert an `SGMLDeclaration` element in the application's definition in `structapps.fm` and type the pathname for the new declaration. For a description of the default SGML declaration and the variations that FrameMaker supports, see Appendix D, "SGML Declaration."

If you need to reapply the parser to a DTD but not recreate the DTD, select File>Structure Tools>Parse Structured Document.

Log files for a translated EDD

If FrameMaker encounters any problems while creating a DTD from an EDD, it produces a log file of warnings and errors. A warning is a notification of a potential problem, but it does not necessarily mean something is wrong with the EDD or the resulting DTD. An error indicates an actual problem in the processing; some errors can cause the processing to stop altogether.

A log file can have warning messages for conditions such as name changes, and it can have error messages for FrameMaker syntax errors, read/write rule errors, capacity and quantity errors, and missing files.

This is an example of a message in a log file:

```
/usr/fmsgml/tutorial/chapter.edd; line 24
Invalid property specified for element "AFrame".
```

The first line in the message gives the location of the problem in the EDD; you can click this line to go to the problem in the EDD. The second line describes the problem; you can click this line to see a longer explanation.

A log file is initially locked so that you can click in it to use the hypertext links. If you want to save the log file, you must first unlock it by pressing Esc Flk. (Press Esc Flk again to relock the file.) For general information on log files, see "Log files" on page 71.

Sample documents and EDDs

FrameMaker comes with several structured documents and EDDs that you can review as samples or use as a starting point for developing EDDs of your own. You'll find the files in the following directories.

- For UNIX:

```
$FMHOME/fminit/uilanguage/Samples/FMSGML/
$FMHOME/fminit/uilanguage/Samples/Templates/Structured/
```

- For Windows:

```
$FMHOME\samples\fmsgml\
$FMHOME\samples\templates\Structured\
```

- For Macintosh:

```
$FMHOME:Samples:FMSGML:
$FMHOME:Samples:Templates:Structured:
```

The `FMSGML` directory contains a commented EDD for this developer's guide and one structured document that uses the EDD.

The `Structured` directory contains structured documents for outlines, reports, and viewgraphs.

9 Structure Rules for Containers, Tables, and Footnotes

Containers, tables, table parts, footnotes, and Rubi groups can all hold other elements; they build the structural hierarchy of a document. For each of these elements in an EDD, you need to define the allowable contents to describe a document structure that is valid.

Content rules are structure rules that describe allowable content in FrameMaker. These rules translate to content models and declared content in markup. If you convert a DTD to an EDD, or an EDD to a DTD, the allowed-content information for elements is preserved. You can modify some of the default translations with read/write rules.

For containers, tables, and some table parts, you can also specify a structure for a new instance of the element in a document. These rules do not describe the range of allowable contents (as content rules do), but provide a starting structure as a convenience for your end users.

In this chapter

This chapter explains how to write structure rules in element definitions for containers, tables, table parts, footnotes, and Rubi groups. In the outline below, click a topic to go to its page.

Summary of required and optional structure rules:

- "Overview of EDD structure rules" on page 110

Syntax of content rules and the translation of content rules to markup:

- "Writing an EDD general rule" on page 111
- "Specifying validity at the highest level in a flow" on page 116
- "Adding inclusions and exclusions" on page 116
- "How content rules translate to markup data" on page 118

Optional rules that specify structure properties for new instances of an element:

- "Inserting descendants automatically in containers" on page 119
- "Inserting table parts automatically in tables" on page 120
- "Inserting Rubi elements automatically in Rubi groups" on page 123

Information to help you correct errors in structure rules:

- "Debugging structure rules" on page 123

Overview of EDD structure rules

All containers, tables, table parts, footnotes, and Rubi groups must have a general rule that specifies what the element is allowed to contain in a document. These elements can also have optional inclusions or exclusions that specify child elements that can or cannot occur in the element and its descendants. The general rule and inclusions and exclusions together make up an element's *content rules*.

> **XML:** The specification for XML does not support inclusions and exclusions. If an EDD uses inclusions and exclusions, when you save the associated FrameMaker file to XML the software ignores them. The result could be a document that is valid in FrameMaker, but invalid in XML. When converting an EDD to a DTD, the resulting XML DTD could contain errors. You should only use inclusions and exclusions in an EDD that is part of an SGML structure application.

At least one container in the EDD must also have a rule stating that the container is valid at the highest level in a structured flow. All other elements in the flow are descendants of this container. For example:

Element (Container): Chapter
General rule: Para+, Section+
Valid as the highest-level element.
Inclusion: CrossRef
Automatic insertions
Automatically insert child: Para

At least one container must have a specification about highest-level validity.

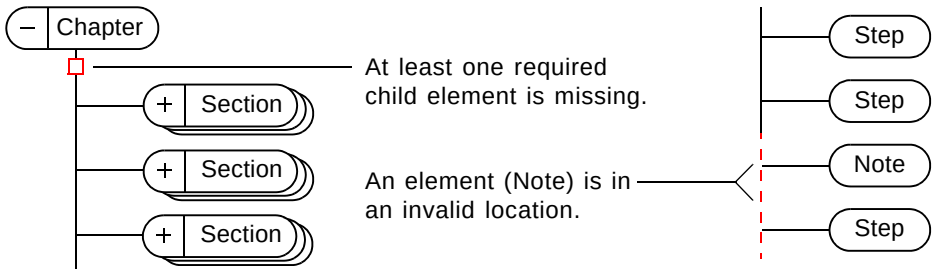
Element (Table): RuleTable
General rule: Title?, Heading, Body, Footing?
Initial structure pattern: Heading, Body

A general rule is required for every container, table, table part, and footnote.

Element (TableBody): Body
General rule: Row+

Element (Footnote): Footnote
General rule: (<TEXT> | Emphasis)+

In a document, if an instance of an element does not conform to the content rules, the Structure View identifies the problem. If an element is missing one or more child elements required in its general rule, a small hole marks the first place where elements are missing. If a child element is in an invalid location according to the general rule and any inclusions or exclusions that apply, the vertical line next to it is broken to the end of the parent. (The hole and dotted line are in red on a color monitor.)



An end user can also validate the document for details on these errors and to find additional errors such as a highest-level element not permitted at that level. For more information on the Structure View and validation, see the FrameMaker user's manual.

Other optional structure rules are available to help you develop a convenient working environment for end users. For containers, you can define descendants to insert automatically along with a new instance of the container in a document. And for tables and most table parts, you can define element tags for child table part elements (in a repeating pattern if necessary).

Writing an EDD general rule

A *general rule* describes the child elements that an element can contain, whether the child elements are required or optional, and the order in which the child elements can occur. It also specifies whether the element can have text. Every element definition for a container, table, table part, footnote, or Rubi group must have a general rule.

When you insert `Container`, `Table`, `Table Heading`, Table `Body`, `Table Footing`, `Table Title`, Table `Cell`, `Table Row`, `Footnote`, `Rubi`, or `Rubi Group` as the element type in a definition, the `GeneralRule` child element is inserted automatically. Type the text of the rule in the `GeneralRule` element. For example:

Element (Container): Section
General rule: Head, Paragraph+, Section*

The tags in the rule must be for elements defined in the current EDD. Containers, footnotes, table titles, and table cells can use any elements in the EDD except for tables and table parts. Tables and other table parts have more specific restrictions on the elements you can use; for information on this, see "Restrictions on general rules for tables" on page 114.

The typographical rules for Rubi groups prohibit line breaks. A Rubi group can use any elements in the EDD, but if you insert a paragraph container element in a Rubi group, FrameMaker will ignore the paragraph break and treat it as a text range. For best results, the general rule for a Rubi group should not include any paragraph container elements.

Syntax of a general rule for EDD elements

A general rule can list child element tags and content symbols, and it can use occurrence indicators, connectors, and parentheses to further describe the contents allowed.

Occurrence indicators and connectors

An occurrence indicator after an element tag specifies whether the child element is required or optional and whether it can be repeated. If you do not use an occurrence indicator, the element is required and can occur once. These are the indicators available:

Symbol	Meaning
Plus sign (+)	Child element is required and can occur more than once.
Question mark (?)	Child element is optional and can occur once.
Asterisk (*)	Child element is optional and can occur more than once.

Multiple element tags in a general rule are separated with connectors that specify the order in which the child elements can occur. These are the connectors available:

Symbol	Meaning
Comma (,)	Child elements must occur in the order given.
Ampersand (&)	Child elements can occur in any order.
Vertical bar (|)	Any one of the child elements in the group can occur.

For example, this general rule specifies that the element must begin with a `Head`, then it must have one or more `Paragraph` elements, and then it can have one or more optional `Section` elements:

```
Head, Paragraph+, Section*
```

This rule specifies that the element can have either one or more `Paragraph` elements or one `List` element:

```
Paragraph+ | List
```

Be careful to write general rules that are not ambiguous. When an end user validates a document, FrameMaker must be able to match child elements to the tags in the general rule without looking ahead to other child elements. For example, this rule is ambiguous because FrameMaker cannot tell whether an `Item` in the document matches the first or second `Item` in the rule without looking ahead for a second `Item`:

```
Item?, Item
```

If you want to specify that an element must have one or two Items, write this rule instead:

```
Item, Item?
```

The connectors in a group of element tags must all be the same type. For example, this rule is erroneous because it uses two different connectors:

```
Caption, Graphic | Table
```

If you need to mix connectors in an element rule, use parentheses to define groups of element tags. In the rule above, if you want a `Caption` followed by either a `Graphic` or a `Table` put parentheses around `Graphic | Table`. For more information, see "Parentheses" on page 113.

Content symbols

A general rule can also use symbols that specify content other than child elements. These are the content symbols available:

Symbol	Meaning
<TEXT>	Element can contain text.
<TEXTONLY>	Element can contain only text. It cannot contain child elements, even inclusions defined in the content rules of its ancestors.
<ANY>	Element can contain any combination of text and elements defined in the EDD.
<EMPTY>	Element cannot contain any text or elements.

You can use the <TEXTONLY>, <ANY>, or <EMPTY> symbol only in a rule by itself, but you can combine the <TEXT> symbol with element tags for more complex expressions. For example, this rule specifies that the element can begin with text and can end with a table:

```
<TEXT>, Table?
```

Text is always optional and repeatable. An occurrence indicator after a token does not change the meaning of the general rule: <TEXT>, <TEXT>+, and <TEXT>* are all equivalent.

Use the <TEXTONLY> token for elements that directly correspond to markup elements with declared content `CDATA` or (for SGML) `RCDATA`.

Use the <EMPTY> symbol for elements you want to remain empty. These are some possible examples of empty container elements:

- A paragraph element that has an autonumber but no content, such as a section number on a line by itself
- A text range element that has a `UniqueID` attribute but no content, to describe a source location within a paragraph for cross-references
- A table cell element that remains empty in tables, such as cells in an empty row to provide space between groups of cells in a table

For information on translation to XML and SGML, see "How content rules translate to markup data" on page 118.

Parentheses

You can use parentheses to group element tags and content symbols in a general rule. The items within a pair of parentheses act as a single tag in the rule's syntax. You can use occurrence indicators and connectors with a group as you do with an individual element tag.

For example, this rule specifies that the element must begin with a Head, then it must have at least one Paragraph or one List element, and then it can have one or more optional Section elements:

Head, (Paragraph | List)+, Section*

Note that because of the plus sign after the parenthesized group, the Paragraph and List elements can be repeated any number of times.

A group can also be nested within another group. For example, this rule specifies that the element must begin with a Front element and then must have either one or more Part elements or one or more Chapter or ErrorSection elements followed by one or more optional Appendix elements:

Front, (Part+ | ((Chapter | ErrorSection)+, Appendix*))

The connectors within a single parenthesized group must be the same, although a group nested within another group can use a different connector.

Make sure that a parenthesized group does not introduce any ambiguities to the general rule. For example, this rule is ambiguous because either alternative can begin with a Preface element:

Preface | (Copyright?, Preface, Foreward)

Outer parentheses around a general rule are optional. If you save an EDD as a DTD, FrameMaker inserts outer parentheses when markup requires it.

Restrictions on general rules for tables

The structural parts of a table each use a general rule to describe how an instance of the table can be built. For example, this general rule specifies that a Table element begins with a Title and then has a Body and either a Heading or a Footing (but not both):

Element (Table): Table
General rule: Title, ((Heading, Body) | (Body, Footing))

The general rule for a table or table part uses the same syntax as the general rule for other elements, but a few additional restrictions apply:

Element type	Restrictions
Table	Limited to one each of the following types of child elements, in this order: title, heading, body, footing. (The rule can specify these elements more than once in Or expressions, but a table element can have only one of each child.) The plus sign (+), asterisk (*), and ampersand (&) are not allowed.

Element type	Restrictions
	The tokens <TEXT>, <TEXTONLY>, <ANY>, and <EMPTY> are not allowed.
Heading, body, or footing	Limited to one or more row child elements.
	The tokens <TEXT>, <TEXTONLY>, <ANY>, and <EMPTY> are not allowed.
Row	Limited to one or more table cell child elements.
	The tokens <TEXT>, <TEXTONLY>, <ANY>, and <EMPTY> are not allowed.
Title or cell	All child elements are allowed, except for tables and table parts.
	The tokens <TEXT>, <TEXTONLY>, <ANY>, and <EMPTY> are allowed.

Note that the table format associated with a table element determines whether a new table has a title. When using an object format rule to assign a format to a table, you need to make sure that the format is consistent with the table's general rule. For more information on these formats, see "Setting a table format" on page 190.

In a document, an end user cannot insert table or table part elements outside their restricted context, even if he or she has all the elements available in the Element Catalog. For example, the user cannot insert a table title after an existing title or a table heading in a container. If the user tries to do this, FrameMaker displays an alert.

Default general rules for EDD elements

If you do not specify a general rule, FrameMaker gives the element a default general rule that depends on the element's type. When you import the EDD into a template, FrameMaker inserts a default general rule wherever the EDD has an empty `GeneralRule` element. You can then save the EDD with these corrections.

These are the default general rules:

Element type	Default general rule
Container	<ANY>
Table	TITLE?, HEADING?, BODY, FOOTING?
Table heading, body, or footing	ROW+
Table row	CELL+
Footnote, table title, or table cell	<TEXT>
Rubi Group	<TEXT>, RUBI
Rubi	<TEXT>

If you try to save an EDD as a DTD when there are empty general rules, the resulting DTD will have syntax errors. You need to import the EDD into a template first so that FrameMaker can insert default general rules for you.

Keep in mind that for an EDD to be valid you need to insert a `GeneralRule` element in the definition for every container, table, table part, footnote, or Rubi group—even when you are not filling in the general rule.

Specifying validity at the highest level in a flow

Every structured flow in a document needs one highest-level element. This element is a container and holds all other elements in the flow. You need to define at least one highest-level element for each type of structured flow that can appear in documents.

To specify validity at the highest level in a flow, insert a `ValidHighestLevel` element right before or after the element's general rule. For example:

Element (Container): Chapter
General rule: Paragraph+, Section, Section+
Valid as highest-level element.

When defining a highest-level element, you may want to give it a name that identifies the type of document or flow. For example, in a chapter document the element might be called `Chapter`.

Provide a highest-level element for book files as well as for flows in document files. For example:

Element (Container): Book
General rule: Front, Chapter+, Index?
Valid as highest-level element.

Adding inclusions and exclusions

XML: The specification for XML does not support inclusions and exclusions. If an EDD uses inclusions and exclusions, when you save the associated FrameMaker file to XML the software ignores them. The result could be a document that is valid in FrameMaker, but invalid in XML. When converting an EDD to a DTD, the resulting XML DTD could contain errors. You should only use inclusions and exclusions in an EDD that is part of an SGML structure application.

An *inclusion* is an element that can occur anywhere inside the defined element or its descendants. Inclusions are often used for elements that might be necessary throughout a hierarchy, such as cross-references or terms with special formatting. An *exclusion* is an element that cannot occur anywhere in the defined element or its descendants.

You can define inclusions and exclusions as part of the content rules for containers, tables, table parts, footnotes, and Rubi groups. Defining inclusions and exclusions in a few high-

level elements saves you the effort of allowing or prohibiting child elements for individual lower-level elements.

Because inclusions and exclusions apply to an element and its descendants, at some point in the hierarchy an element may be both included and excluded. When this happens, FrameMaker does not allow the element to occur. For example, an `Index` element might be specified as an inclusion in a `Report` element. If a `Head` element excludes `Index` elements, the `Index` is excluded even though `Head` is a descendant of `Report`.

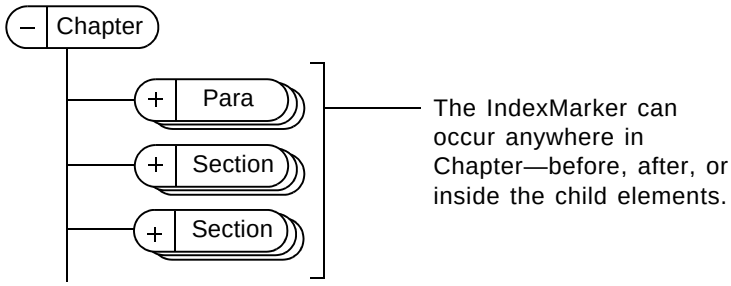
Inclusions

To add an inclusion to an element definition, insert an `Inclusion` element anywhere after the general rule (or optional validity specification) and before the format rules. Then type the tag of the element you want to include.

For example, instead of adding `IndexMarker` to the general rules for `Para` and `Section` (and all the elements they contain), you might specify `IndexMarker` as an inclusion for `Chapter`:

Element (Container): Chapter
General rule: Para+, Section*
Valid as highest-level element.
Inclusions: IndexMarker

In the example, an `IndexMarker` can occur before or after a `Para` or `Section` as well as anywhere within a `Para` or `Section` (unless one of the elements specifies `IndexMarker` as an exclusion):



When defining an inclusion, look for descendants that should not use the inclusion. Add an exclusion in the descendants' definitions to prohibit the inclusion in those contexts.

If you want more than one inclusion, for each additional inclusion, insert an `Inclusion` element and type the tag, or put multiple element names in the same element, separated by commas.

An inclusion can use any element tag defined in the current EDD. An end user will be able to insert the included element only if it is allowed in the context, even though it may be a valid inclusion. For example, the user cannot insert a table footnote between table rows

even though the footnote may be a valid inclusion in the table because table footnotes are allowed only in titles and cells.

Exclusions

To add an exclusion to an element definition, insert an `Exclusion` element anywhere after the general rule (or optional validity specification) and before the format rules. Then type the tag of the element you want to exclude.

For example, you might use an exclusion to prevent end users from creating nested `Procedure` elements:

Element (Container): Procedure
General rule: Step+
Exclusions: Procedure

The most common uses of exclusions are to prevent nesting and to counter an inclusion for a particular context.

If you want more than one exclusion, for each additional exclusion, insert an `Exclusion` element and type the tag, or put multiple element names in the same element, separated by commas.

How content rules translate to markup data

In FrameMaker, the general rule and the inclusions and exclusions use a syntax that is based on SGML model groups and declared content. (The occurrence indicators, connectors, and parentheses are the same in both environments.) On import or export between an EDD and a DTD, the content information about child elements is preserved. Note that you do not need to put parentheses around the entire general rule in FrameMaker.

When you convert an EDD to a DTD, FrameMaker also translates content symbols in general rules:

- The FrameMaker content symbol `<TEXT>` translates to a content token of `#PCDATA`. (`#PCDATA` can be combined with element tags, as `<TEXT>` can be in FrameMaker.)
- The FrameMaker general rule `<TEXTONLY>` translates to declared content of `RCDATA`.
- The FrameMaker general rule `<ANY>` translates to the reserved name `ANY` in a markup content model.
- The FrameMaker general rule `<EMPTY>` translates to a declared content of `EMPTY`.

When you convert a DTD to an EDD, FrameMaker performs the translations in reverse. In addition, a declared content of `CDATA` translates to `<TEXTONLY>` in FrameMaker.

For more detailed information on how content rules translate to XML or SGML, see Chapter 16, “Translating Elements and Their Attributes.” For information on how structured tables translate to XML or SGML, see Chapter 18, “Translating Tables.”

Inserting descendants automatically in containers

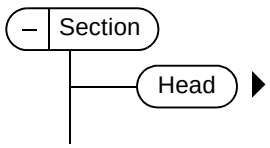
In the definition of a container, you can specify nested descendants to insert automatically. Whenever an end user inserts the container in a document, the descendants are inserted automatically along with it. This makes it convenient for users to work with containers that always begin with the same structure.

To insert descendants automatically, insert an `AutoInsertions` element anywhere after the general rule (or optional validity specification) and before the format rules. For the first descendant, insert an `InsertChild` element and type the element tag. Then for each additional descendant, insert an `InsertNestedChild` element and type the tag. You can have one `InsertChild` element and any number of `InsertNestedChild` elements.

For example, this autoinsertion rule specifies that a new `Section` begins with a nested `Head`:

Element (Container): Section
General rule: Head, Para+
Automatic insertions
Automatically insert child: Head

If the last descendant in the autoinsertion sequence is a container that allows text, the insertion point is placed automatically inside the container, ready for the end user to add text:

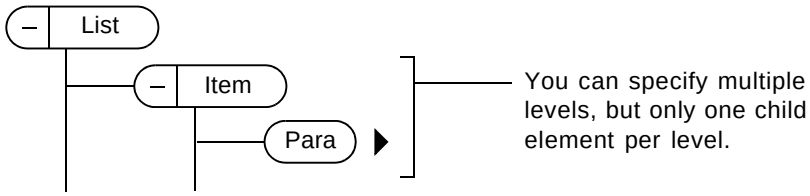


FrameMaker cannot insert sibling elements automatically. In the `Section` and `Head` example above, even though a `Section` may always begin with a `Head` and then a `Para`, you can have only the `Head` inserted automatically and not a `Para` too.

You can, however, have FrameMaker insert a sequence of descendants, in which each nested descendant has one child element inserted along with it. For example, this autoinsertion rule specifies that a new `List` has a nested `Item`, and the `Item` has a nested `Para`:

Element (Container): List
General rule: Item+
Automatic insertions
Automatically insert child: Item
and nested child: Para

This is the structure of the new `List`:



Keep in mind that the autoinsertion rules from one element definition do not carry over to another. In the `List` and `Item` example above, even though the `Item` definition in the same EDD may specify `Para` as a nested descendant, to insert `Para` automatically with `List` you still need to explicitly specify `Para` (and `Item`) in the `List` definition.

The descendants in autoinsertion rules do not need to be containers. If you use a table, graphic, cross-reference, variable, marker, footnote, or equation as a nested descendant, that element should be the last one in the sequence of descendants. The automatic insertion stops after the non-container element is inserted.

When FrameMaker inserts descendants automatically in a document, it opens dialog boxes as necessary.

- If a descendant is a table, graphic, cross-reference, variable, marker, or equation and FrameMaker requires information about the element (such as a table format or a graphic filename), the appropriate dialog box opens as the element is inserted.

The end user can cancel a dialog box, and the autoinsertion stops at that point. (This does not affect any descendants that were already inserted automatically.) Be aware of the dialog boxes that may open during autoinsertion so that you can design a behavior that is reasonable for the user.

Inserting table parts automatically in tables

All tables have certain characteristics in common: A table has at least a body; a table heading, body, or footing has at least one row; and a table row has the same number of cells as there are columns in the table. Whenever an end user inserts a structured table, or part of one, FrameMaker automatically inserts the necessary child elements to build a basic structure.

You can define an initial structure pattern for a table, heading, body, footing, or row, and FrameMaker will use that structure when an end user inserts the table or table part in a document. If you do not define a structure pattern, FrameMaker gives new instances of the element a default initial structure it derives from the general rule.

When an end user inserts a new table element, he or she uses the Insert Table dialog box to specify the number of heading, body, and footing rows and the number of columns. The initial structure pattern (or the general rule) determines which row elements to use in the heading, body, and footing, and which cell elements to use in the columns.

Initial structure pattern

To define an initial structure pattern for a table, heading, body, footing, or row, add an `InitialStructurePattern` element anywhere after the general rule and before the format rules. Then list the tags of child elements in the initial structure, separated by commas.

For a table element, the initial structure pattern must include at least a body and can have at most a title, a heading, a body, and a footing, in that order. When FrameMaker gives a table its structure, it uses each type of table part in the pattern. For example, this pattern specifies a table with an initial structure of `Heading` and `StandardBody`:

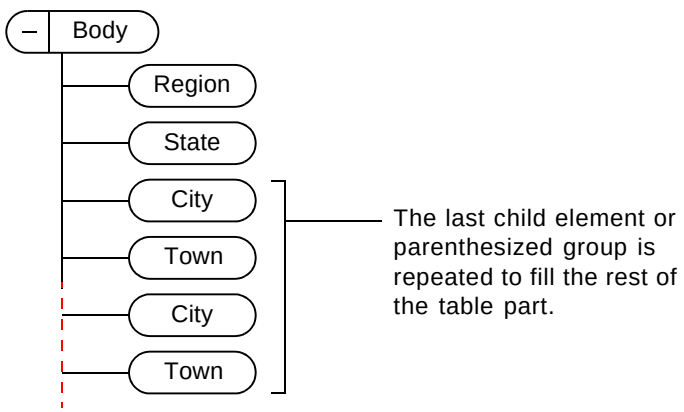
Element (Table): Table
General rule: Title?, Heading, (RulesBody | StandardBody)
Initial structure pattern: Heading, StandardBody

A table format specifies whether or not a new table has a title. Although you can include a title in the initial structure pattern, this information is overridden by the table format. For information on giving a table element a format, see "Setting a table format" on page 190.

For a heading, body, footing, or row, if the table part needs more elements than appear in the structure pattern, FrameMaker repeats the last item in the syntax to fill the number of rows and cells required. (In these cases, the defined structure is truly a pattern.) The last item may be a single child element or a parenthesized group. For example, this pattern specifies a body using the row elements `Region`, `State`, `City`, and `Town`:

Element (TableBody): Table
General rule: Region, State, City, Town, Zip
Initial structure pattern: Region, State, (City, Town)

If an end user inserts a table with this body and specifies six body rows in the Insert Table dialog box, the new table has the following body row structure:



It is also possible that not all of a structure pattern will be used. For example, if an end user inserts a table with the body defined above and specifies two body rows, the body in the new table has only a `Region` row and a `State` row.

If an end user adds a row to an existing table by inserting a row element, the structure of the new row is based on the initial structure pattern of the parent heading, body, or footing. (But if the user adds a row by pressing Control-Return or by using the Add Rows or Columns command, the new row takes the structure of the row with the insertion point.)

The syntax of an initial structure pattern is restricted as follows:

- The only arguments allowed are element tags. The content symbols <TEXT>, <TEXTONLY>, <EMPTY>, and <ANY> are not allowed.
- The comma (,) is the only connector allowed. No occurrence indicators are allowed.
- Parentheses are allowed to group element tags.

For more information on the symbols allowed, see "Restrictions on general rules for tables" on page 114.

Be sure that the child elements in an initial structure pattern are valid according to the general rule of the table or table part. Otherwise, a new table may have invalid structure (such as in the example above with `City` and `Town`).

Default initial structure

If you do not specify an initial structure pattern in an element definition, FrameMaker uses the general rule to give a new table or table part its initial structure.

For a table element, FrameMaker builds a default initial structure by taking the first of each type of table part in the table's general rule. For example, suppose a table element does not have an initial structure pattern but has the following general rule, where `Normal` and `Special` are two types of table bodies:

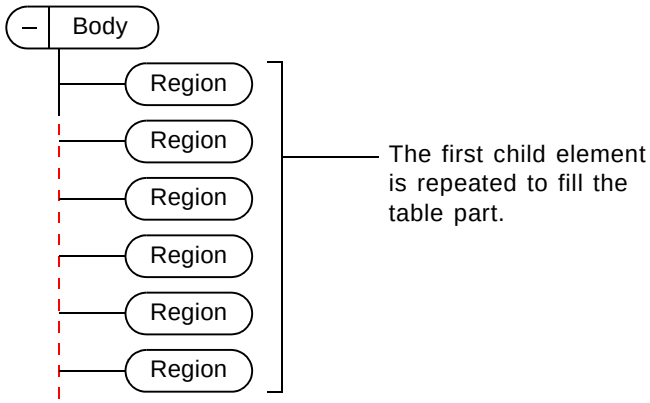
Element (Table): Table
General rule: Title?, Heading, (Normal | Special)

If an end user inserts a table, uses a table format that includes a title, and specifies some heading and body rows, the new table has an initial structure of `Title`, `Heading`, and `Normal`. (If the user also gives the table some footing rows, those rows have the default tag `Footing`.)

For a table heading, body, footing, or row, the default initial structure is simply the first element tag to appear in the general rule. If the table part needs more child elements, the first child element is repeated to fill the number of rows and cells required. For example, suppose a body element does not have an initial structure pattern but has the following general rule, where `Region`, `State`, and `City` are rows:

Element (TableBody): Body
General rule: Region, State, City

If an end user inserts a table with this body and specifies six body rows in the Insert Table dialog box, the new table has the following row structure:



Inserting Rubi elements automatically in Rubi groups

A Rubi group always includes a Rubi element as its last child. Whenever an end user inserts a Rubi group, FrameMaker automatically inserts the necessary child elements to build a basic structure.

You can define an initial structure pattern for a Rubi group and FrameMaker will use that structure when an end user inserts the Rubi group in a document. If you do not define an initial structure pattern, FrameMaker gives new instances of the Rubi group a first Rubi element found in the Rubi group's general rule. If no Rubi element is specified in the general rule, FrameMaker inserts a default Rubi element named `RUBI`.

Initial structure pattern

For a Rubi Group element, the initial structure pattern can specify one child Rubi element. To define an initial structure pattern for a Rubi group, add an `InitialStructurePattern` element anywhere after the general rule and before the format rules. Then specify the tag of the child Rubi element. For example, this pattern specifies a Rubi group with an initial child Rubi element named `MyRubiElement`:

Element (Rubi Group): MyRubiGroup
General rule: Oyamoji, (MyRubiElement | SpecialRubiElement)
Initial structure pattern: MyRubiElement

Debugging structure rules

After writing structure rules, you should try them out by importing the EDD into a sample document and inserting, wrapping, and moving a variety of elements. If any elements that you expect to be valid are displayed as invalid (or vice versa) in the Structure View, check the EDD for these errors:

- Typing errors in element tags, content symbols, or symbols in any structure rule
- A general rule that has ambiguous element tags or mixed connectors (For advice on avoiding these problems, see page 113.)

- Inclusions or exclusions specified for an element that is not a container or for a container with the general rule <TEXTONLY>
- An element used at the highest level in its flow that does not have a specification for highest-level validity

If FrameMaker identifies any problems when you import an EDD, it produces a log file of warnings and errors. For information on how to work with this file, see "Log files for imported element definitions" on page 104.

To help you locate ambiguous element tags or mixed connectors, the log file shows a caret symbol (^) in front of a tag or a connector that may be incorrect. For example, this ambiguous general rule is displayed as follows in the log file:

^Item?, ^Item

10 Text Format Rules for Containers, Tables, and Footnotes

You can define text formatting properties for containers, tables, table parts, and footnotes in FrameMaker. When an end user inserts one of these elements in a document, text in the element or in a descendant is formatted automatically according to the format rules in the EDD.

Text formatting in FrameMaker is hierarchical—an element can inherit its properties from ancestors and pass on its properties to descendants.

XML uses CSS and XSL to express formatting of a document apart from the markup data. When reading XML, FrameMaker does not use the CSS data or any of the formatting expressed in XSL. Instead, FrameMaker uses the template associated with the XML structure application to format the data. When writing XML, FrameMaker can generate a CSS file based on the formatting in the document template. You can also use the File > Structure Tools > Generate CSS2 command to generate a CSS.

The formatting information in the template includes format definitions as well as format rules specified in the EDD. You specify whether or not to generate the CSS in the `structapps.fm` file. For more information, see "Managing CSS generation" on page 62 and Chapter 15, "Saving EDD Formatting Information as a CSS Stylesheet."

Markup does not provide a mechanism for formatting text, so the text format rules you write in FrameMaker do not have counterparts in markup. If you import a DTD, you can add format rules to the resulting EDD for use in FrameMaker. If you import a markup document, the text in the document is formatted for elements that have format rules in the EDD you use. If you export a document or EDD to markup, the text formatting information is not preserved.

In this chapter

This chapter explains how to write text format rules for containers, tables and their parts, and footnotes in FrameMaker. In the outline below, click a topic to go to its page.

Background on text format rules and inheritance:

- "Overview of text format rules" on page 126
- "How elements inherit formatting information" on page 127

Syntax of text format rules:

- "Specifying an element paragraph format" on page 132
- "Writing context-dependent format rules" on page 132

- "Defining the formatting changes in a rule" on page 143
- "Specifications for individual format properties" on page 146

Other format rules for special purposes:

- "Writing first and last format rules" on page 156
- "Defining prefixes and suffixes" on page 158
- "When to use an autonumber, prefix or suffix, or first or last rule" on page 162

Format change lists you can refer to and limits on values in change lists:

- "Defining a format change list" on page 163
- "Setting minimum and maximum limits on properties" on page 165

Information to help you correct errors in format rules:

- "Debugging text format rules" on page 166

Overview of text format rules

The text format rules in an element definition can have any combination of the following:

- A reference to a "base" paragraph format stored in the document. The paragraph format defines all aspects of text and paragraph formatting—including font properties, indentation, line spacing, alignment, autonumbering, and hyphenation properties. If a definition does not have a reference to a format, the defined element inherits its format from an ancestor.
- One or more format rules that describe changes to the paragraph format in use. A change can apply to the element in all contexts where it appears or only in a particular context, as specified in each rule. Several rules can apply in one context.

  Format rules can list changes to specific formatting properties, or they can refer to a different paragraph format, to a character format (if you're formatting the element as a text range), or to a list of changes stored elsewhere in the EDD.

When FrameMaker formats text in an element, it applies a paragraph format to the element along with any appropriate changes described in format rules. For example, a Head element might have a different point size at different section levels, but in other respects it would be formatted the same everywhere. If you anticipated having two levels for the element, you could define its formatting in this way:

Element (Container): Head
General rule: <TEXT>
Text format rules
Element paragraph format: head — The base paragraph format for all Head elements
1. **Count ancestors named:** Section
If level is: 1
No additional formatting.
Else, if level is: 2
Default font properties
Size: 14pt

If a Head appears in a second-level Section, the format rule changes the point size to 14.

By using context-dependent format rules, you don't need to define and maintain a separate paragraph format for each place in which an element can occur.

Any part of an element's formatting information can be inherited from an ancestor's definition. For example, you might want to indent a `Section` element and its descendants when it is nested within another `Section`. You could specify the change in indentation once, in a format rule for `Section`, and the descendants of `Section` would inherit this information:

Element (Container): Section
General rule: Head, (Para | List)+, Section*
Text format rules
1. **If context is:** Section
Basic properties
Move left indent by: +0.5"

If a Section is nested, the Section and its descendants are indented .5 inch for each level of nesting.

To write text format rules that are easy to maintain, you should normally define as little formatting information as possible in each element definition and let the elements inherit whatever properties they share with their ancestors. Using inheritance judiciously can greatly simplify your format rules.

The only table-part elements that can contain text are table titles and cells. Although you can write format rules for table, heading, body, footing, and row elements, in these cases the rules specify text formatting only for their descendant titles and cells.

In a document, if an end user applies a different paragraph format to an element or applies formatting changes to an element, the changes are considered format rule overrides. When the user re-imports element definitions, he or she can either leave the overrides as they are or remove the overrides so that the formatting in elements with text conforms to the text format rules.

How elements inherit formatting information

In a typical document, many elements can not only use the same paragraph format but also share changes to the format. Text formatting in FrameMaker is *hierarchical*, which means that elements can inherit all or part of their formatting information from ancestors. This lets you control common formatting information in parent elements.

It is even possible to have only one paragraph format for an entire document. The format is associated with the document's highest-level element, and all other elements inherit the format and specify changes to it when necessary.

The general case

When FrameMaker formats text in an element, it first determines which paragraph format to apply:

- If the element's definition specifies a base paragraph format, that format is used.
- If the element's definition does not specify a paragraph format, FrameMaker searches up through the element's ancestors until it finds an element with a format and then uses that format.

  In general, if FrameMaker reaches the top of the element's hierarchy and still has not found a format, it uses the default `Body` paragraph format for the document. (The behavior is somewhat different for an element in a table or footnote, or if the document is part of a book. See "Inheritance in a table or footnote" on page 130 or "Inheritance in a document within a book" on page 131.)

After FrameMaker takes a paragraph format from an ancestor or from the top of the hierarchy, it starts at that point and goes back down through the hierarchy to the current element, picking up formatting changes in format rules along the way. The changes modify the paragraph format cumulatively at each point. A format can have an *absolute value* (a fixed value, such as an indent expressed as distance from the left margin) or a *relative value* (a change to a current setting, such as an amount to move an indent). An absolute value replaces the same value in the paragraph format, and a relative value is added to the format's value to create a new value.

For example, in this set of definitions only the `Section` element has a base paragraph format (`body`). The descendants of `Section` all use the `body` paragraph format, and most also specify changes to it:

Element (Container): Section
General rule: Head, (Para | List)+, Section*
Text format rules
Element paragraph format: body
1. **If context is:** Section
Basic properties
Move first indent by: +0.5"
Move left indent by: +0.5"

This paragraph format applies to Section and its descendants.

Element (Container): Head
General rule: <TEXT>
Text formal rules
1. **In all contexts**
Default font properties
Weight: Bold
Size: 14pt

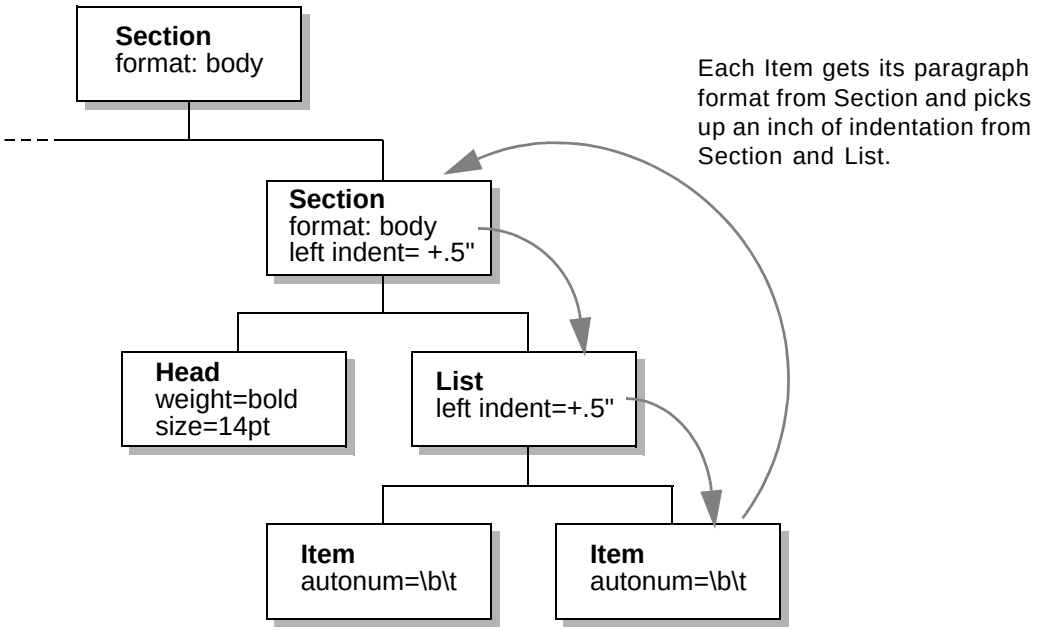
Element (Container): Para
General rule: <TEXT>

Element (Container): List
General rule: Item+
Text formal rules
1. **In all contexts**
Basic properties
Move left indent by: +0.5"

Element (Container): Item
General rule: <TEXT>
Text formal rules
1. **In all contexts**
Numbering properties
Autonumber format: \b\t

When no paragraph format is specified, the default format `Body` is used. Note that because case is significant in format tags, `body` in the example is not the same as the tag `Body`.

The following document structure uses these definitions. FrameMaker formats text in the `Item` elements by searching up to the second-level `Section` for a paragraph format. The `Item` elements are left-indented .5 inch more by the format rule for the second-level `Section` and another .5 inch by the `List` element's format rule. They also have a bullet provided by an autonumber in their own definition.



Note that when FrameMaker picks up changes from ancestors, it includes changes from the element with the paragraph format.

Inheritance in a table or footnote

If the current element is in a table or a footnote, FrameMaker searches through the element's ancestors for a paragraph format in the same way that it does for other elements, with these differences:

- For text in a table title or table cell, FrameMaker does not search beyond the ancestor table element. If it reaches the table element and still has not found a paragraph format, it uses the paragraph format stored in the table format being used for the appropriate type of table part.
- For text in an element in a footnote, FrameMaker does not search beyond the ancestor footnote element. If it reaches the footnote element and still has not found a paragraph format, it uses the document's current footnote paragraph format (if the footnote is in the main flow) or current table-footnote paragraph format (if the footnote is in a table).

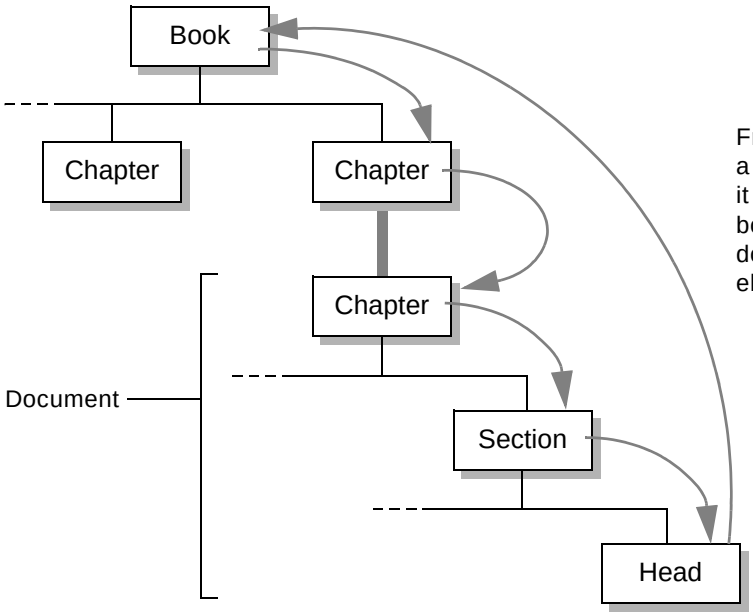
  The Paragraph Catalog in a document stores default `Footnote` and `TableFootnote` paragraph formats, and an end user can also define custom footnote paragraph formats. FrameMaker applies whichever format is specified in the document's Footnote Properties or Footnote Table Properties dialog box. For information on footnote paragraph formats, see the FrameMaker user's manual.

Inheritance in a document within a book

In a document that is part of a book, FrameMaker searches through an element's ancestors for a paragraph format in the same way that it does in other documents. If FrameMaker does not find a format in the document (when searching outside a table or footnote), it continues looking for one in the ancestor elements of the book file.

If FrameMaker finds a paragraph format in the book file, it uses that format. If it reaches the top of the element's hierarchy in the book and still has not found a format, it uses the default Body paragraph format stored in the document.

When FrameMaker uses a format from an ancestor in a book or from the top of the hierarchy, it starts at that point and then goes back down through the hierarchy to the current element, picking up formatting changes along the way.



FrameMaker can search for a format in a book file. Then it picks up changes in the book and brings them back down to the current element.

Only a document's main flow is considered to be part of a book's hierarchy. FrameMaker continues the search up into the book file if the current element is part of the main flow.

When an end user updates a book, the element hierarchy from the book file is stored in the book's documents. To apply a paragraph format, FrameMaker looks at the book's hierarchy in the document and then uses the appropriate format from the document's format rules. Closing or even deleting the book file does not affect the book information in a document; a user can regenerate the book without the document to remove the information.

If the end user adds a document to the book, the book's text format rules are not automatically applied to elements in the document. The user needs to update the book again.

Specifying an element paragraph format

An element definition can include a reference to a base paragraph format. If an instance of the element contains text, the text's format is the paragraph format plus any changes specified for the current context in the element's format rules. A paragraph format is also passed on to the element's descendants—until a descendant provides a different format.

To specify an element paragraph format, insert an `ElementPgfFormatTag` element as the first child element of `TextFormatRules`, and type the tag of the format. The tag must refer to a paragraph format stored in the Paragraph Catalog of the documents. For example, this element uses the `item` paragraph format, and a format rule specifies an autonumber format that is either a bullet or an incrementing number (in either case, followed by a tab):

Element (Container): Item
General rule: <TEXT>
Text format rules
Element paragraph format: item ——— The base paragraph format
1. If context is: BulletList
Numbering properties
Autonumber format: \b\t
Else, if context is: NumberList
Numbering properties
Autonumber format: <n+>\t

Usually, paragraph formats do not vary greatly from one element to another. Your definitions will be simpler and the catalogs easier to maintain if you specify paragraph formats for as few elements as necessary and let most elements inherit their format. You can use format rules to handle changes to the format for particular elements. For information on inheritance, see "How elements inherit formatting information" on page 127.

Table, heading, body, footing, and row elements cannot contain text themselves, but can hold other elements that do. If you specify a paragraph format for one of these elements, the format applies only to the element's descendants that can contain text.

A paragraph format can also be part of a context-dependent format rule so that it is used only in some cases. This is described in "Writing context-dependent format rules," next.

Writing context-dependent format rules

You can define changes to the element paragraph format in use with one or more format rules. The rules describe possible contexts in which an element can occur and give formatting changes for each context. When FrameMaker formats text in the element, it uses the current paragraph format (which may have been inherited and modified by ancestors' definitions), plus any format rules that apply to the current context.

A format rule provides context and formatting information:

- The rule can apply to all contexts in which the element occurs, or it can define particular contexts or the number of levels deep the element is nested in an ancestor. If the rule

defines contexts or levels, it can have separate if, else/if, and else clauses for different possibilities.

- Each “in all contexts” rule or if, else/if, and else clause specifies formatting changes. The formatting changes can be a list of specific properties; or a reference to a different paragraph format, a character format (if you are formatting the element as a text range), or a list of properties stored elsewhere in the EDD.

For example:

Element (Container): Head
General rule: <TEXT>
Text format rules
1. **In all contexts.**
Default font properties
Weight: Bold
2. **If context is:** Section < Section
Numbering properties
Autonumber format: <n>.<n+>\t
Else, if context is: Section
Numbering properties
Autonumber format: <n+>\t

A format rule clause has a context specification …

… and formatting changes for that context.

A format rule or clause can also include a context label to help end users select elements when inserting cross-references or preparing a table of contents or other generated list. An element can have more than one format rule, and format rules can be nested inside one another. The format rules are numbered automatically.

This section describes the context specifications and context labels in format rules. For a summary of the formatting changes available, see “Defining the formatting changes in a rule” on page 143.

An element’s format rules or paragraph format can also be inherited from an ancestor. For information on this, see “How elements inherit formatting information” on page 127.

All-contexts rules

A format rule can specify a formatting change that applies to an element in all contexts in which it can occur. To write an all-context format rule, insert an `AllContextsRule` element, and then define the formatting changes for the rule.

In this example, the text in a `TableHead` element is in 14-pt boldface no matter where the element occurs in a document:

Element (Table Head): TableHead
General rule: TableHeadRow+
Text format rules
1. **In all contexts.**
Default font properties
Size: 14pt
Weight: Bold

Context-specific rules

A format rule can define one or more possible contexts, with formatting changes for each context. The contexts are expressed in separate if, else/if, and else clauses. When applying a format rule to an element, FrameMaker uses the first clause in the rule that is true for the instance of the element. (Clauses in other rules may also apply.)

To write a context-specific format rule, insert a `ContextRule` element. An `If` element and a nested `Specification` element are inserted automatically along with it. (The `Specification` element does not have a label in the document window.) Type one or more element tags to define the `If` context, and then define the formatting changes for that context. If you need additional clauses in the format rule, you can insert and define any number of `ElseIf` elements, ending with one `Else` element.

Defining a context

When defining a context, you can name the parent element or a list of ancestors. For a list of ancestors, begin with the parent and then name successively higher-level ancestors, separating the element tags with a less-than sign (<).

In this example, an `Item` begins with a bullet if it occurs in a `List` within a `Preface`, or with an incrementing number if it occurs in a `List` within a `Chapter`:

Element (Container): Item
General rule: <TEXT>
Text format rules
1. **If context is:** List < Preface
Numbering properties
Autonumber format: \b\t
Character format: bulletsymbol
Else, if context is: List < Chapter
Numbering properties
Autonumber format: <n+>\t

The ancestors in a list can also be instances of the same element, to describe nesting within that element. For example, this specification is true if an element's parent is a `Section` that is a child element of another `Section`:

Section < Section

Note that a nesting specification of this type is true whenever the current element is nested in *at least* as many levels as shown in the rule. That is, `Section < Section` applies a

formatting change if the current element is nested within two or more `Section` elements. (For a way to describe nesting that means *exactly* the level indicated, see "Level rules" on page 138.)

Wildcards for ancestors

Use an asterisk (*) as a wildcard to represent an unspecified number of successive ancestors in the hierarchy. For example, this specification is true if an element's parent is a `Section` and any one of the parent's ancestors is also a `Section`:

```
Section < * < Section
```

OR indicators

Use OR indicators (|) to test the specification for any ancestor in a group. Separate the element tags of the ancestors with an OR indicator, and enclose the group in parentheses. For example, this specification is true if an element appears in a `List` within a `Preface` or a `Chapter`:

```
List < (Preface | Chapter)
```

Sibling indicators

Use a sibling indicator to describe an element's location relative to its siblings. You can use the indicator to describe the relationship of the current element to its siblings or of an ancestor element to its siblings. Enclose the sibling indicator in braces ({ }).

To describe the relationship of the current element to its siblings, type the sibling indicator and a less-than sign, and then continue with the parent and other ancestors. For example, this specification is true if an element is the first element in its parent `NumberList`:

```
{first} < NumberList
```

To describe the relationship of an ancestor to its siblings, append the sibling indicator to the ancestor's tag. For example, this specification is true if an element's parent is a `Section` in a `Chapter` and the `Section` immediately follows a `Title`:

```
Section {after Title} < Chapter
```

You can also use a sibling indicator by itself as the entire context specification. For example, this specification is true if an element is the only child element in its parent:

```
{only}
```

These are the sibling indicators you can use:

Indicator	Specification is true if the element is
`{first}`	The first element in its parent
`{middle}`	Neither the first element nor the last element in its parent
`{last}`	The last element in its parent
`{notfirst}`	Not the first element in its parent
`{notlast}`	Not the last element in its parent

Indicator	Specification is true if the element is
{only}	The only element in its parent
{before *sibling*}	Followed by the named element or text content
{after *sibling*}	Preceded by the named element or text content
{between *sibling1*, *sibling2*}	Between named elements or text content
{any}	Anywhere in its parent (equivalent to no indicator)

The *sibling* argument with the before, after, and between indicators can be an element tag or the keyword <TEXT>. If you use <TEXT>, FrameMaker looks to see if the element is preceded or followed by text rather than by a sibling element. A string generated by a prefix, suffix, or autonumber is not considered to be text.

Rather than defining a first or last relationship between siblings, you may want to write a first or last format rule for the parent. See "Writing first and last format rules" on page 156.

Attribute indicators

You can also use an attribute name/value pair in a text format rule clause to more narrowly define context. For the context specification to be true, an instance of the element must have the attribute name and value specified.

If the element does not have an attribute value but the attribute is defined to have a default value, the default value is used. If the attribute does not have a default value, you can test for an attribute that has no value by specifying an empty string (*attr* ="") for the attribute value.

To test an attribute of the current element, type the attribute name and value in brackets as the context specification. To use an attribute with an ancestor, type the attribute name and value in brackets after the ancestor tag. Separate the attribute name and value with an equal sign, and enclose the value in double quotation marks. If the specification has a sibling indicator, put the attribute information before the indicator.

For example, this rule specifies that an Item begins with a bullet if it occurs in a List that has a Type attribute with the value Bullet, or it begins with an incrementing number if it occurs in a List that has a Type attribute with no specified value:

Text format rules
1. **If context is:** List [Type = "Bullet"]
   Numbering properties
   Autonumber format: \b\t
   Character format: bulletsymbol
   Else, if context is: List [Type = ""]
   Numbering properties
   Autonumber format: <n+>\t

By using attributes in format rules, you may be able to define fewer elements than you would need to otherwise. With the List example above, you can have just one definition

for List and rely on attribute values to determine whether an instance of List is bulleted or numbered. Without the attributes, you would need to define separate elements for bulleted and numbered lists.

You can type straight or curved quotation marks around the attribute values (they are automatically curved if you have Smart Quotes on in Text Options). If you need to type a double quotation mark as part of a value, escape the mark with a backslash (\).

To use a set of attribute name/value pairs, separate the pairs with an ampersand (&). For the specification to be true, an instance of the element must have all of the attribute pairs. For example, this specification is true if the element's parent is List and the element has a Type attribute with the value Numbered and a Content attribute with the value Procedure:

List [Type = "Numbered" & Content = "Procedure"]

These are the operators you can use in attribute name/value pairs:

Operator	With attributes of
= (equal to)	All types
!= (not equal to)	All types
> (greater than)	Choice and numeric types
< (less than)	Choice and numeric types
>= (greater than or equal to)	Choice and numeric types
<= (less than or equal to)	Choice and numeric types

The definition for a Choice attribute includes a list of possible values. If you use a greater-than sign or a less-than sign with a Choice attribute in a format rule, FrameMaker evaluates the name/value pair using the order in the list of values, with the "lowest value" being the one on the left. For example, this pair specifies any Security value that is to the left of Classified in the defined list for the Security attribute:

Report [Security < "Classified"]

With the numeric attributes, you can also express a range of values by combining attribute name/value pairs that use a greater-than sign or a less-than sign. For example, these pairs specify an inclusive range from 12 to 20:

Note [Width >= "12" & Width <= "20"]

For information on defining attributes, see Chapter 16, "Translating Elements and Their Attributes."

Order of context clauses

When a context-specific format rule has more than one clause, keep in mind that FrameMaker applies the first clause in the rule that is true for the instance of the element. You must write rule clauses from the most specific to the most general.

For example, suppose you want to apply a formatting change to an `Item` when it appears inside a nested `List` element (a `List` inside a `List`). If you put the context specifications for the `Item` in the following order, FrameMaker would never apply the second clause because an `Item` in a nested `List` also matches the first specification:

List
List < List

You get the effect you want by reversing the clauses.

Level rules

When defining the nesting depth of an element within levels of another element, you may find it easier to use a *level rule* rather than a normal context rule. In a level rule, you name the ancestor to the current element and then in each clause count the number of times the ancestor appears above the current element.

For example, suppose you're describing the nesting depth of a `Head` in `Section` elements. This is how you would define it using a normal context rule:

Element (Container): Head
General rule: <TEXT>
Text format rules
1. **If context is:** Section < Section < Section
   Default font properties
   Font size: 12
   Else, if context is: Section < Section
   Default font properties
   Font size: 14
   Else, if context is: Section
   Default font properties
   Font size: 18

This is the same specification using a level rule instead:

Text format rules
1. **Count ancestors named:** Section
   If level is: 1
   Default font properties
   Font size: 18
   If level is: 2
   Default font properties
   Font size: 14
   If level is: 3
   Default font properties
   Font size: 12

Note that in context rules you need to go from the lowest level to the highest, but in level rules the order of clauses is arbitrary so you can go from highest to lowest if you prefer. In

context rules `Section < Section < Section` means "nested in *at least* three `Section` elements," and in level rules a `Section` count of 3 means "nested in *exactly* three `Section` elements." Because the level counts are exact, you can specify them in any order.

To write a level rule, insert a `LevelRule` element. In the `CountAncestors` element that is inserted automatically, type the element tag of the ancestor to count. Insert an `If` element (a nested `Specification` element is inserted automatically along with it), and type the level number of the ancestor. Then define the formatting changes for that level.

If you need additional clauses in the format rule, you can insert and define any number of `ElseIf` elements, ending with one `Else` element. Each specification needs a level number and formatting changes for that level. An `Else` clause includes a count of 0 unless there is an `If` or `ElseIf` clause for that case.

Level rules are usually simpler than context rules (especially with three or more levels), because you do not need to spell out the context in each clause.

You can name a list of ancestors in a level rule, and FrameMaker will count any ancestor in the list. Separate the ancestors with a comma. For example, this specification is true if an element is nested within the specified number of levels of any combination of `Section` and `Chapter` elements:

Section, Chapter

You cannot mix context clauses and level clauses together in a single format rule, though you can nest a context rule in a level clause or a level rule in a context clause, and one element definition can have both kinds of rules.

Using the current element in the count

A level rule can also count instances of the current element in the hierarchy. You do not need to provide the element's tag as ancestor information. After inserting the `LevelRule` element, delete the `CountAncestors` element that is inserted automatically and begin with the `If` specification. The current instance of the element is included in the count.

For example, this rule specifies that a `Section` is indented .5 inch if it is nested within another `Section`:

Element (Container): Section
General rule: Head, Para+
Text format rules
1. If level is: 2
Basic properties
Indents
Move left indent by: +0.5"

The rule in this example is also applied to any `Head` and `Para` descendants of the `Section`.

Stopping the count at an ancestor

You can have FrameMaker stop counting when it reaches a particular element in the hierarchy. This allows you to use different formatting changes for a nested element in different contexts. Insert the `StopCountingAt` element after the `CountAncestors` element, and type the tag of the element to stop counting at. Then continue with the `If`, `ElseIf`, and `Else` specifications.

For example, suppose you want list items in a table cell to be formatted based on the nesting of lists in the cell. If the entire table can occur in a list in the document, you want to test for the nesting level of lists only in the table, and not in the overall nesting of lists in the document. You do this by counting lists in successive ancestors until you reach a table. In the following rules, an `Item` in a second-level `List` can be indented .5 inch or 1 inch. The `Item` is indented .5 inch if the nested `List` is in a `Table`:

Element (Container): Item
General rule: <TEXT>
Text format rules
1. **Count ancestors named:** List
   Stop counting at first ancestor named: Table
   If level is: 2
   Basic properties
   Move left indent by: +0.5"
2. **Count ancestors named:** List
   Stop counting at first ancestor named: Chapter
   If level is: 2
   Basic properties
   Move left indent by: +1.0"

For the first rule to be true, the hierarchy between the current `Item` and the closest `Table` ancestor must include two `List` elements. For the second rule to be true, the hierarchy between the `Item` and the `Chapter` ancestor must include two `List` elements.

Nested format rules

Within any format rule clause, you can nest another entire format rule. By nesting rules, you can sometimes organize rules more efficiently and make them easier to read. Look for places in a format rule where context information is repeated, and see if you can combine all the formatting information for that context in a nested rule.

To add a nested format rule, insert a `Subrule` element after the context specification for the outer rule. Specify the nested rule in the same way that you write a main rule.

For example, suppose an `Item` element can occur in either a numbered list or a bulleted list, and in a numbered list the autonumber format is different for the first position in the list than for other positions. You might write the format rule in this way:

Element (Container): Item
General rule: <TEXT>
Text format rules
1. **If context is:** {first} < List[Type="Numbered"]
Numbering properties
Autonumber format: <n=1>\t
Else, if context is: List[Type="Numbered"]
Numbering properties
Autonumber format: <n+>\t
Else
Numbering properties
Autonumber format: \b\t

These context specifications can be combined.

Rather than having two rule clauses for the numbered-list contexts, you can have one main clause with a nested rule (using a `SubRule` element) that describes the two variations:

Text format rules
1. **If context is:** List[Type="Numbered"]
1.1 If context is: {first}
Numbering properties
Autonumber format: <n=1>\t
Else
Numbering properties
Autonumber format: <n+>\t
Else
Numbering properties
Autonumber: \b\t

The outer rule determines whether the context is a numbered list.

If it is a numbered list, the inner rule determines which autonumber format to use.

You can nest a level rule in a context clause, or a context rule in a level clause.

Multiple format rules

An element definition can have more than one format rule. Like nesting rules, writing separate format rules can help you organize information efficiently. Look for places in a format rule where formatting changes are repeated, and see if you can break out the changes into a separate rule.

To add another format rule for an element, insert an `AllContextsRule`, `ContextRule`, or `LevelRule` element after an existing rule, and specify the context and formatting changes. Write the format rules in the order you want them to be interpreted.

For example, suppose a `Head` element has the same font change in any context but a different autonumber format in different nesting levels. This shows the formatting changes described in a single rule:

Element (Container): Head
General rule: <TEXT>
Text format rules
1. **If context is:** Section < Section
Default font properties
Weight: Bold
Numbering properties
Autonumber format: <n>.<n+>\t
Else:
Default font properties
Weight: Bold
Numbering properties
Autonumber format: <n+>\t

These font properties can be combined.

Rather than repeating the `Bold` specification, you can break out that part into a separate rule for all contexts:

Text format rules
1. **In all contexts.**
Default font properties
Weight: Bold
2. **If context is:** Section < Section
Numbering properties
Autonumber: <n>.<n+>\t
Else:
Numbering properties
Autonumber: <n+>\t

The first rule sets the weight to Bold for every Head element.

Keep in mind that FrameMaker applies format rules in the order they appear in the definition, so it is possible for a format rule to override an earlier rule. For example, if rule 1 changes the weight of text in all contexts and rule 2 applies a different paragraph format in certain contexts, the paragraph format overrides the change in the font weight.

Context labels

In some dialog boxes, FrameMaker displays a list of element tags for an end user to select from. For example, a user selects element tags in the Set Up dialog box (Generate command) to set up a generated file such as an index or a table of contents.

You may want FrameMaker to distinguish among instances of some elements in these lists. For example, an end user might include `Head` elements in a table of contents when the parent of the `Head` is a first- or second-level `Section`, but not when the parent is a more deeply nested `Section`. To allow FrameMaker to distinguish element instances, you provide a context label with formatting variations of the element.

In a dialog box, the end user sees an element once for each context label it can have and once for all contexts in which no label applies. The user selects an element with a label to work with all instances of the element from the context associated with the label.

To provide a context label in a format rule clause, insert a `ContextLabel` element after the `If`, `ElseIf`, or `Else` specification, and type the text of the label. A context label cannot contain white-space characters or any of these special characters:

```
( ) & | , * + ? < > % [ ] = ! ; : { } "
```

For example, this rule has a context label for different levels of `Head` elements:

Element (Container): Head
General rule: <TEXT>
Text format rules
1. **Count ancestors named:** Section
If level is: 1
Context label: 1st-level
Default font properties
Font size: 18
Else, if level is: 2
Context label: 2nd-level
Default font properties
Font size: 14

In a dialog box that lists element tags, the end user sees the `Head` element like this:

Head(<no label>)
Head(1st-level)
Head(2nd-level)

You can use the same label in more than one clause or format rule. Context labels are not inherited by descendants.

If multiple labels apply to an instance of an element (because of multiple rules with labels that apply), only the last appropriate label in the definition is stored with the instance. Thus, only the last label determines whether the instance appears in a generated list.

Defining the formatting changes in a rule

Each clause in a text format rule includes formatting changes that apply to the element in the specified context. The changes can modify paragraph properties or text-range properties, or they can specify no additional formatting for the context.

Paragraph formatting

You can define paragraph-formatting changes for any container, footnote, table, or table part. Insert a `ParagraphFormatting` element after the context specification, and then specify one of these changes:

- Refer to a paragraph format stored in the document. A paragraph format is a fully specified set of properties—including font settings, indentation, line spacing, alignment, and autonumbering. The format replaces the paragraph format for this context only. (It does not become the base format for the element and is not inherited by descendants.)

Insert the element ParagraphFormatTag, and type the tag of the paragraph format.

- Make changes to particular paragraph format properties. The changes modify the specified properties in the paragraph format in use.

  Insert the element corresponding to the group of properties (such as PropertiesFont), and define the changes. For a summary of these properties, see "Specifications for individual format properties" on page 146.

- Refer to a list of changes to properties stored elsewhere in the EDD. The changes in the list modify the specified properties in the paragraph format in use.

  Insert the element FormatChangeListTag, and type the tag of the change list. For information on defining a list, see "Defining a format change list" on page 163.

For example:

Element (Container): Head
General rule: <TEXT>
Text format rules
1. **If context is:** {first} < (Chapter | Appendix)
   Use paragraph format: chaptitle
   Else
   Use paragraph format: head
2. **Count ancestors named:** Section
   If level is: 1
   No additional formatting.
   Else, if level is: 2
   Default font properties
   Size: 14pt

Text range formatting

A *text range* in FrameMaker is a string of text within a paragraph; it often has different font properties from the paragraph text around it. Common examples of text ranges are emphasized phrases, book titles, and code fragments. You can format a container as a text range to apply font changes and to override or replace paragraph properties.

Insert a TextRangeFormatting element after the context specification, and then specify one of these changes:

- Refer to a character format stored in the document. The character format replaces the font properties in the paragraph format for the element in this context.

  Insert the element CharacterFormatTag, and type the tag of the format.

- Make changes to particular font properties. The changes modify the specified font properties in the paragraph format in use.

  Insert the PropertiesFont element, and define the changes. For information on the properties, see "Font properties" on page 150.

- Refer to a list of changes to properties stored elsewhere in the EDD. The changes in the list modify the specified font properties in the paragraph format in use.

  Insert the element `FormatChangeListTag`, and type the tag of the change list. For information on defining a list, see "Defining a format change list" on page 163.

For example:

Element (Container): Emphasis
General rule: <TEXT>
Text format rules
1. In all contexts.
Text range.
Font properties
Angle: Oblique

A text range label appears when you format a container as a text range.

Element (Container): CodeFragment
General rule: <TEXT>
Text format rules
1. In all contexts.
Text range.
Use format change list: Code

If you use a format change list with a text range, only the font properties from the list are applied.

No additional formatting

You can also specify no formatting changes, and the element in that context will be formatted according to the current paragraph format (which may have been inherited and modified by ancestors' format rules). Insert a `NoAdditionalFormatting` element after the context specification.

The no-formatting option provides a way to express "if not" in a multiple-clause format rule. For example, this rule specifies that if a `Head` does *not* appear in a `LabeledPar` or `Sidebar` element, the formatting changes are made as described:

Element (Container): Head
General rule: <TEXT>
Text format rules
1. If context is: * < (LabeledPara | Sidebar)
No additional formatting.
Else
1.1. Count ancestors named: Section
If level is: 1
No additional formatting.
Else, if level is: 2
Default font properties
Size: 12pt
Weight: Bold

The formatting changes apply if a Head does not appear in this context.

This clause is not necessary but makes the nested rule (1.1) more readable.

In some cases, you may also want to use the no-formatting option to improve the readability of a format rule. In the nested rule (1.1) in the example above, you could leave out the `If level is: 1` clause and begin with `If level is: 2`. But the level-1 information makes the rule more comprehensive and will remind anyone reading the rule later that the paragraph format in use is correct for `Head` elements in a first-level `Section`.

Specifications for individual format properties

A format rule clause can describe changes to any property available in the Paragraph Designer (except for Next Paragraph Tag, which is not used in structured documents). These properties include indentation, alignment, line spacing, font and style settings, autonumbering, paragraph straddle formats, and hyphenation limits.

The design properties in a rule clause are organized in the groups Basic, Font, Numbering, Pagination, Advanced, and Table Cell—analogous to the layout of these properties on pages in the Paragraph Designer. To specify a set of properties, insert the element that corresponds to the group as a child element in the `ParagraphFormatting` or `TextRangeFormatting` element, and then proceed with the individual properties. For example:

Element (Container): Head
General rule: <TEXT>
Text format rules
1. **If context is:** Section < Section
Basic properties
Paragraph spacing
Change space above by: +2pt
Default font properties
Weight: Bold
PairKerning: Yes
Numbering properties
Autonumber format: <n>.<n+>\t

Properties are organized in groups analogous to pages in the Paragraph Designer.

With some properties you type the value, and with others you use a keyword child element (such as `Bold` or `Yes`) to express the value. Most of the numeric values you can enter are absolute, but a few values in the Basic, Font, and Table Cell groups are relative. A relative value can be positive or negative. (The plus sign is not required with a positive value.) You cannot enter both an absolute value and a relative value for a property.

An absolute value overrides the current value for the property in the paragraph format in use. A relative value is added to the current value to set a new value. For example, suppose an element inherits a paragraph format with a left indent of .25 inch:

- With a `LeftIndent` value of 1 inch, the new left indent is 1 inch.
- With a `LeftIndentChange` value of 1 inch, the new left indent is 1.25 inch.

You can specify a unit of measure, such as `pt` or `in`, for indentation and tab stops in the Basic properties and the frame position in the Advanced properties; if you do not specify a

unit, FrameMaker uses the one set in View Options in the document. Offsets and spreads in Font properties and word spacing in Advanced properties are always in percentages. For any other numeric value, if you specify a unit it is converted to points.

These are the possible units of measure:

Unit	Notation in the EDD
Centimeters	`cm`
Millimeters	`mm`
Inches	`"` or `in`
Picas	`pc`, `pi`, or `pica`
Points	`pt` or `point`
Didots	`dd`
Ciceros	`cc` or `cicero`
Percentage	`%`

Properties that can use relative values have minimum and maximum limits. Font sizes, for example, must fall within the inclusive range 2 to 400 points. If you set a value that is outside an allowed range (either by specifying an absolute value or by calculating a new relative value), FrameMaker changes the value to the minimum or maximum. You can also set your own minimum and maximum limits. For a summary of the minimum and maximum limits and information on changing them, see "Setting minimum and maximum limits on properties" on page 165.

For more details on the formatting properties and guidelines on how to use them, see the FrameMaker user's manual.

Basic properties

The Basic properties set indentation, line spacing, paragraph alignment, paragraph spacing, and tab stops. To begin changing these properties, insert a `PropertiesBasic` element in the `ParagraphFormatting` element.

The numeric settings for Basic properties all allow an absolute value or a relative value. Use the elements with `Change` in the tag for relative values.

Indentation, spacing, and alignment

The following elements and values define the properties for indentation, spacing, and alignment:

`Indents` — Distances from the left and right edges of the text column to the text.

`FirstIndent` *`dimension`* Sets the left indent for the first line in a paragraph. Type the distance from the left edge of the text column.

	`FirstIndentRelative` *`dimension`* Sets the left indent for the first line in a paragraph (relative to the left indent in use). Type a relative value. `FirstIndentChange` *`dimension`* Sets the left indent for the first line in a paragraph (added to the current first indent). Type a relative value. `LeftIndent` *`dimension`* Sets the left indent for all lines in a paragraph after the first line. Type the distance from the left edge of the text column. `LeftIndentChange` *`dimension`* Sets the left indent for all lines in a paragraph after the first line (added to the current left indent). Type a relative value. `RightIndent` *`dimension`* Sets the right indent for all lines in a paragraph. Type the distance from the right edge of the text column. `RightIndentChange` *`dimension`* Sets the right indent for all lines in a paragraph (added to the current right indent). Type a relative value.
`LineSpacing`	Vertical space between lines in a paragraph, measured from baseline to baseline. (See the note on font size and line spacing after this table.) `Height` *`dimension`* Sets the space between lines in a paragraph. Type the distance from one baseline to the next. `HeightChange` *`dimension`* Sets the space between lines in a paragraph (added to the current line spacing). Type a relative value. `Fixed` Keeps line spacing the same everywhere in a paragraph. `NotFixed` Allows line spacing in a paragraph to change to accommodate the largest font on each line.
`PgfAlignment` *`keyword`*	Sets left, center, right, or justified for the horizontal position of a paragraph within the left and right indents. Insert a keyword child element from the catalog to specify the type of alignment.
`ParagraphSpacing`	Vertical space above and below a paragraph, measured from the baseline of the first and last lines. `SpaceAbove` *`dimension`* Sets the space above a paragraph. Type the distance above the baseline of the first line.

`SpaceAboveChange` *`dimension`* Sets the space above a paragraph (added to the current space above). Type a relative value.

`SpaceBelow` *`dimension`* Sets the space below a paragraph. Type the distance below the baseline of the last line.

`SpaceBelowChange` *`dimension`* Sets the space below a paragraph (added to the current space below). Type a relative value.

When you change a font size in a document or in an EDD, FrameMaker automatically recalculates the paragraph's line spacing for you. (In a single-spaced paragraph, the default spacing is 120 percent of the font size.) You can change a font size and specify your own line spacing for it rather than using the calculated spacing. When doing this in an EDD, note the following behavior:

- If you change the font size and the line spacing in a single rule clause, the spacing is changed to the value you set, overriding any value that FrameMaker calculates for the new font size.
- If you change the font size and the line spacing in two different rule clauses, the line spacing clause must come *after* the font size clause for your spacing to override the calculated value. (If the font size comes second, FrameMaker calculates the spacing at that point and overrides the value you set earlier.)

For determining the space between two adjacent paragraphs, FrameMaker uses the space below the first paragraph or the space above the second paragraph, whichever is larger. If the paragraph is at the top of a column, the `SpaceAbove` value is ignored; at the bottom of a column, the `SpaceBelow` value is ignored.

Tab stops

To set tab stops for an element, insert a `TabStops` element in `PropertiesBasic` and then continue with the following elements and values. A single format rule clause can have one or more `TabStop` elements *or* one `MoveAllTabStopsBy` element *or* one `ClearAllTabStops` element.

`TabStop` Definition of one tab stop.

`TabStopPosition` *`dimension`* Sets the location that the insertion point jumps to when an end user presses Tab. Type the distance from the left edge of the text column to the tab stop.

`RelativeTabStopPosition` *`dimension`* Sets the location that the insertion point jumps to when an end user presses Tab. Type a relative value from the left indent of the paragraph to the tab stop. A positive value moves the tab stop right; a negative value moves it left.

	`TabAlignment` *`keyword`* Sets left, center, right, or decimal alignment for tabbed text on a tab stop. Insert a keyword child element from the catalog to specify the type of alignment. `AlignOn` *`character`* For decimal tabs, specifies a character on which to align the tabs. Type one character (usually a period or a comma). `Leader` *`string`* Specifies characters to repeat in a line to create a leader between a tab and the character following it. Type one or more characters (can be any characters including spaces).
`MoveAllTabStopsBy` *`real-number`*	Changes the positions for all inherited tab stops (added to the current tab stop positions). Type a relative value. A positive value moves the tab stops right; a negative value moves them left.
`ClearAllTabStops`	Removes all inherited tab stops.

Font properties

The Font properties set the font, size, and style of text in an element. To begin changing these properties, insert a `PropertiesFont` element in the `ParagraphFormatting` or `TextRangeFormatting` element. In a `ParagraphFormatting` element, the label you see in the EDD is `Default font properties`; in `TextRangeFormatting`, the label is `Font properties`.

Most of the Font properties use a child element to set the value, and in many cases FrameMaker inserts a default child element for you. You can select the child element and change it to a different one if you need to.

The following elements and values define the Font properties:

`Angle` *`keyword`*	Sets a font angle (such as italic or cursive). Insert a keyword child element to specify the angle.
`Case` *`keyword`*	Sets a capitalization style (such as lowercase or small caps). Insert a keyword child element to specify the case.
`ChangeBars` *`boolean`*	If Yes, displays a vertical line in the page margin where an end user has made changes to text.
`Color` *`name`*	Specifies a color for text. Type the name of a color defined in Color Definitions in the document.
`CombinedFont` *`name`*	Specifies a combined font. Type the name of the combined font that is available to your users. Note that the combined font must be defined in the FrameMaker document that will use this EDD.
`Family` *`name`*	Specifies a typeface family (such as Times or Helvetica). Type the name of a font that is available to your end users.

`OffsetHorizontal` *real-number*	Moves a text range element. Type the percentage of an em space you want the element to move. (FrameMaker interprets a value of 20 as 20% of an em space.) A positive value moves the element right; a negative value moves it left.
`OffsetVertical` *real-number*	Moves a text range element. Type the percentage of an em space you want the element to move. (FrameMaker interrupts a value of 20 as 20% of an em space.) A positive value moves the element up; a negative value moves it down.
`Outline` *boolean*	If `Yes`, applies an outline style to text (only on a Macintosh).
`Overline` *boolean*	If `Yes`, places a line over text.
`PairKerning` *boolean*	If `Yes`, turns on ligatures and moves character pairs closer together as necessary to improve appearance. The ligatures, character pairs, and amount of kerning depend on the font.
`Shadow` *boolean*	If `Yes`, applies a shadow style to text (only on a Macintosh).
`Size` *dimension*	Sets a point size for text. Type a number of points from 2 to 400. (The notation `pt` or `point` is optional.) (See the note on font size and line on page 149.)
`SizeChange` *real-number*	Sets a point size for text (added to the current point size). Type a relative value in points. (The notation `pt` or `point` is optional.) (See the note on font size and line on page 149.)
`Spread` *real-number*	Renamed to `Tracking` (see below). `Spread` will still work, for backward compatibility.
`SpreadChange` *real-number*	Renamed to `TrackingChange` (see below). `SpreadChange` will still work, for backward compatibility.
`Stretch` *real-number*	Sets the amount to stretch or compress the characters. Type a percentage of the font's em space. (The symbol % is optional.) A positive value stretches the characters; a negative value compresses them. Normal stretch is 0 percent.
`StretchChange` *real-number*	Sets the amount to stretch or compress the characters (added to the current stretch). Type a relative value as a percentage of the font's em space. (The symbol % is optional.)
`Strikethrough` *boolean*	If `Yes`, places a line through text.
`Superscript` `Subscript` *keyword*	Changes characters to a script above or below the baseline. Insert a keyword child element to specify superscript or subscript. The text size and the amount of offset are determined by Text Options in the document.

`Tracking` *`real-number`*	Sets the space between characters. Type a percentage of the font's em space. (The symbol % is optional.) A positive value increases the spread; a negative value decreases the spread. Normal spread is 0 percent.
`TrackingChange` *`real-number`*	Sets the space between characters (added to the current tracking). Type a relative value as a percentage of the font's em space. (The symbol % is optional.)
`Underline` *`keyword`*	Sets an underline style (such as double underline or numeric underline). Insert a keyword child element to specify the style.
`Variation` *`keyword`*	Sets a font variation (such as compressed or expanded). Insert a keyword child element to specify the variation.
`Weight` *`keyword`*	Sets a font weight (such as bold or black). Insert a keyword child element to specify the weight.

Pagination properties

The Pagination properties affect the placement of a paragraph on a page and determine how to break the paragraph across columns and pages. To begin changing these properties, insert a `PropertiesPagination` element in the `ParagraphFormatting` element.

The following elements and values define the Pagination properties:

`KeepWithNext` *`boolean`*	If `Yes`, keeps a paragraph in the same text column as the next paragraph.
`KeepWithPrevious` *`boolean`*	If `Yes`, keeps a paragraph in the same text column as the previous paragraph.
`Placement`	Placement of a paragraph on a page. `AcrossAllCols` Makes a paragraph straddle the width of all columns in its text frame. `AcrossColsSideHeads` Makes a paragraph straddle the width of the columns and the side-head area. `InColumn` Keeps a paragraph in its text column so that it does not straddle other columns. `RunInHead` Displays a paragraph as a head that runs into the next paragraph. A run-in head can have default punctuation. `SideHead` Displays a paragraph as a side head across from the next paragraph. A side head can have default punctuation. `DefaultPunctuation` *`string`* For a run-in head or a side head, specifies punctuation to appear after the head. Type one or more characters (can be any characters including spaces).

	`Alignment` *`keyword`* For a side head, aligns the baseline of the head with the first baseline, top baseline, or top edge of the paragraph across from it. Insert a keyword child element to specify the type of alignment.
`StartPosition`	Location in a text column or page where a paragraph always begins. `Anywhere` Starts a paragraph right below the preceding one. The paragraph's widow/orphan setting determines where it breaks across text columns. `TopOfColumn` Starts a paragraph at the top of the next text column. `TopOfLeftPage`, `TopOfPage`, or `TopOfRightPage` Starts a paragraph at the top of the next specified page.
`WidowOrphanLines` *`integer`*	Sets the minimum number of lines in a paragraph that can appear alone at the top or bottom of a text column. Type a value from 0 to 100.

Numbering properties

The Numbering properties specify the syntax and format for an automatically generated string, such as a number that appears at the beginning of a procedure step. To begin changing these properties, insert a `PropertiesNumbering` element in the `ParagraphFormatting` element. (An autonumber can also be a fixed text string, such as the word *Note* at the beginning of a paragraph.)

If a paragraph element with an autonumber also has a prefix or suffix, the prefix appears just after the autonumber at the beginning of the paragraph, and the suffix appears just before the autonumber at the end of the paragraph.

To read about the syntax of autonumbers and the building blocks available, see the FrameMaker user's manual.

The following elements and values define the Numbering properties:

`AutonumberFormat` *`syntax`*	Specifies the style and incrementing of automatically generated numbers or bullets. Type text and building blocks to define the syntax, or insert building block child elements.
`AutonumCharFormat` *`tag`*	Applies a character format to an autonumber. Type the tag of a character format stored in the document.
`NoAutonumber` *`boolean`*	If `Yes`, turns off autonumbering for the element.
`Position` *`keyword`*	Displays an autonumber at either the beginning of its paragraph or the end of its paragraph. Insert a keyword child element to specify the position.

Advanced properties

The Advanced properties set hyphenation and word spacing options and determine whether to display a graphic with a paragraph. To begin changing these properties, insert a PropertiesAdvanced element in the ParagraphFormatting element.

The following elements and values define the Advanced properties:

FrameAbove *name*	Displays a graphic from a reference frame above a paragraph. Type the name of a frame stored on a reference page in the document. You can also set a position for the frame.
FrameBelow *name*	Displays a graphic from a reference frame below a paragraph. Type the name of a frame stored on a reference page in the document. You can also set a position for the frame.
FramePosition *boolean*	Specifies whether the left alignment of the frame above or the frame below matches the text column or the paragraph indent. Yes specifies that it matches the paragraph indent.
Hyphenation	Hyphenation properties for all text in an element. Hyphenate *boolean* if Yes, turns on automatic hyphenation for the element. Language *string* specifies the language to use for hyphenation rules. MaxAdjacent *integer* Sets the maximum number of consecutive lines that can end with a hyphen. Type a value. ShortestPrefix *integer* Sets the minimum number of letters in a word that can precede a hyphen. Type a value. ShortestSuffix *integer* Sets the minimum number of letters in a word that can follow a hyphen. Type a value. ShortestWord *integer* Sets the minimum length of a hyphenated word. Type a value.
WordSpacing	The amount or word spacing that FrameMaker can decrease or increase in an element. These settings are percentages of the standard word spacing for the font. Normal spacing is 100 percent. Values below 100 allow tighter spacing; values above 100 allow looser spacing. LetterSpacing *boolean* If Yes, allows additional space between characters in justified text to keep the space between words from going over the maximum. Maximum *integer* Sets the largest space allowed between words before FrameMaker hyphenates words or adds letter spacing in justified paragraphs. Type a percentage of the font's em space. (The symbol % is optional.)

`Minimum` *`integer`* Sets the smallest space allowed between words. Type a percentage of the font's em space. (The symbol `%` is optional.)

`Optimum` *`integer`* Sets the optimum amount of space between words. Type a percentage of the font's em space. (The symbol `%` is optional.)

Table Cell properties

The Table Cell properties customize the margins of cells and the vertical alignment of text in them. To begin changing these properties, insert a `PropertiesTableCell` element in the `ParagraphFormatting` element.

A margin or alignment you set with these properties overrides the default properties for a table. For information about how custom margins and alignments work with table formats, see the FrameMaker user's manual.

The following elements and values define the Table Cell properties:

`CellMargins` Margins between the borders of a cell and the text in the cell.

`Bottom`, `Left`, `Right`, or `Top` Specifies the margin to change. For each margin, you can set a custom value or a value that is relative to the table format's margin.

`Custom` *`dimension`* Type the distance in points from the border of the cell to the text. (The notation `pt` or `point` is optional.)

`FromTblFormatPlus` *`dimension`* Type a relative value in points that is added to the default margin set in the table format. (The notation `pt` or `point` is optional.)

`VerticalAlignment` *`keyword`* Sets top, middle, or bottom alignment for text in a table cell. Insert a keyword child element from the catalog to specify the type of alignment.

Asian Text Spacing properties

Typographic rules for variable width Asian text require specifications for spacing between Asian characters, Western and Asian characters, and for punctuation characters. To begin changing these properties, insert a `PropertiesAsianSpacing` element in the `ParagraphFormatting` element.

These properties only take effect when displaying a document on a system running Asian system software.

The following elements and values define the Asian text spacing properties:

`WesternAsianSpacing` The spacing between pairs of Western and Asian characters.

	`Minimum`, `Maximum`, and `Optimum` Specifies the range of spacing percentages to use, and the optimum, or preferred, percentage of spacing to use.
`AsianAsianSpacing`	The spacing between pairs of Asian characters.
	`Minimum`, `Maximum`, and `Optimum` Specifies the range of spacing percentages to use, and the optimum, or preferred, percentage of spacing to use.
`Punctuation`	`Float`, `Fixed`, `Monospace` Specifies the type of character squeezing to use for punctuation characters.

Writing first and last format rules

You can apply a special set of format rules to the first and last paragraphs in an element. This is particularly useful for parent elements that are only containers for child elements that are formatted as paragraphs (often all of the same type), such as a `List` that contains `Item` elements or a `Chapter` that contains `Section` elements.

To write first or last format rules, insert a `FirstParagraphRules` or `LastParagraphRules` element at the same level as `TextFormatRules`, and define the context and formatting specifications as you do for other format rules. For information on the internal specifications of rule clauses, see "Writing context-dependent format rules" on page 132 and "Defining the formatting changes in a rule" on page 143.

For example, these first and last format rules display a line from a reference page above and below a `List` element:

Element (Container): List
General rule: Head?, Item+
Text format rules
1. **In all contexts.**
Basic properties
Indents
Move left indent by: +12pt
Format rules for first paragraph in element
1. **In all contexts**
Advanced properties
Frame above: SingleLine
Format rules for last paragraph in element
1. **In all contexts**
Advanced properties
Frame below: SingleLine

The first and last rules display a line above and below a List.

Note that if the first and last paragraphs are child elements, you can get the same first and last formatting using `{first}` and `{last}` sibling indicators in format rules for the child elements. The first and last format rules are more natural with the hierarchical model, however. Because the properties are associated with the parent, they are inherited by any

paragraph that happens to be the first or last child—so you need to define the special formatting only once. (In the `List` example above, with `{first}` and `{last}` indicators, you would need to define the lines for both the `Head` and the `Item` child elements.)

There are some similarities between the uses of first and last format rules, autonumber strings, and prefixes and suffixes. For guidelines on using the different constructs, see "When to use an autonumber, prefix or suffix, or first or last rule" on page 162.

How first and last rules are applied

The first or last format rules in an element apply to the first or last paragraph in the element. The paragraphs may be child elements, or they may just be text formatted as paragraphs. The rules are ignored in contexts in which a first or last child element is formatted as a text range.

When FrameMaker formats an element in a document, it applies format rules in this order:

- The element's text format rules
- The element's first or last format rules
- The element's prefix or suffix format rules
- The text format rules in any child elements

If an element has a prefix formatted in a separate paragraph, the prefix is the first paragraph. Similarly, a suffix formatted in a separate paragraph is the last paragraph.

If the element has no text content, no child elements, and no prefix or suffix as a separate paragraph, the first or last rules apply to the element itself.

A first or last rule with an autonumber

A first or last rule can be used with an autonumber to display the number or string with only the first or last paragraph in the element. For example, this rule displays the string *Important:* or *Note:* at the beginning of the first `Para` in `Note`:

Element (Container): Note
General rule: Para+
Format rules for first paragraph in element
1. **If context is:** [Important = Yes]
   Numbering properties
   Autonumber format: Important:
   Character format: Bold
   Else
   Numbering properties
   Autonumber format: Note:
   Character format: Bold

If an element contains a single paragraph, and the first and last rules both specify an autonumber, only the first rule is used.

A first or last rule can also apply to a prefix or suffix that is formatted in a paragraph of its own. Because the formatting part of prefix and suffix rules allows only specification of font changes, you can use a first or last rule to give the prefix or suffix special paragraph formatting properties. For an example of this, see "A prefix or suffix for a sequence of paragraphs" on page 160.

Defining prefixes and suffixes

A *prefix* is a text range defined in the EDD that appears at the beginning of an element (before the element's content); a *suffix* is a text range that appears at the end of an element (after the content). In many cases, a prefix or suffix is formatted differently than other text in the element.

You can define a prefix or suffix for any container element in FrameMaker, using a set of rules similar to other format rules. Prefix and suffix rules describe both the text string and any special font properties for it.

To define a prefix or suffix, insert a `PrefixRules` or `SuffixRules` element at the same level as `TextFormatRules`, and write one or more rules for the prefix or suffix. In one of the rules, you need to specify a text string, using the `Prefix` or `Suffix` child element. The string can include any characters (including tabs and spaces) and one or more attribute building blocks.

If you want to include a left angle bracket (<) in the string, escape the bracket with a backslash, like this: \<. (A left angle bracket by itself begins an attribute building block.)

The format rules for a prefix or suffix can use the same context specifications available for other format rules and any of the font changes for text ranges. For information on the internal specifications of rule clauses, see "Writing context-dependent format rules" on page 132 and "Text range formatting" on page 144.

You can display a prefix or suffix with element text in a cross-reference or in a running header or footer in a document. Use the `$elemtext` (rather than the `$elemtextonly`) building block in the cross-reference format or in the header or footer definition. For more information, see the FrameMaker user's manual.

There are some similarities between the uses of first and last format rules, autonumber strings, and prefixes and suffixes. For guidelines on when to use the different constructs, see "When to use an autonumber, prefix or suffix, or first or last rule" on page 162.

How prefix and suffix format rules are applied

The format rules for a prefix or suffix describe font changes only for the prefix or suffix. The changes do not apply to descendants of the element.

> **Important:** Font changes for a prefix/suffix do not take effect if the prefix/suffix begins with a forced return.

When FrameMaker formats an element with a prefix or suffix, it applies format rules in this order:

- The element's text format rules
- The element's first or last format rules
- The element's prefix or suffix format rules

If the element has first or last rules and the prefix or suffix is formatted in a paragraph of its own, the prefix or suffix is the first or last paragraph for the purposes of formatting. Because of the order that rules are applied, a prefix or suffix format rule can override font changes in a first or last format rule.

A prefix or suffix for a text range

You can use a prefix or suffix to provide a string for a text range element inside a paragraph. For example, suppose you want double quotation marks to appear around the text of a quotation every time, without the end user having to type in the marks. You can set up a pair of quotation marks for the `Quotation` text range element as a prefix and a suffix:

Element (Container): Quotation
General rule: <TEXT>
Text format rules
1. **In all contexts.**
Text range.
No additional formatting.

Prefix rules
1. **In all contexts.**
Prefix: “

Suffix rules
1. **In all contexts.**
Suffix: ”

In this example, the prefix and suffix do not have any font changes, so they are formatted the same as other text in the `Quotation` element.

If you want fixed text to appear at the beginning or end of a paragraph rather than inside the paragraph, define the string as a prefix, suffix, or autonumber for the paragraph element. For information on autonumbers, see “Numbering properties” on page 153.

A prefix or suffix for a paragraph

If you define a prefix for a paragraph element that begins with text, the prefix appears at the beginning of the paragraph; if you define a suffix for an element that ends with text, the suffix appears at the end of the paragraph. (In these cases, a prefix for suffix is similar to an autonumber for the paragraph.) For example, you might use a prefix to display *Important:* at the beginning of a paragraph:

Element (Container): Note
General rule: <TEXT>
Prefix rules
1. **In all contexts.**
Prefix: Important:

If a paragraph element also has an autonumber, the prefix appears after an autonumber at the beginning of the paragraph, and a suffix appears before an autonumber at the end of the paragraph.

A prefix or suffix for a sequence of paragraphs

You can define a prefix or suffix for a paragraph element that contains other paragraphs. If the element begins with a paragraph child element, the prefix appears in a paragraph of its own; if the element ends with a paragraph child, the suffix appears in a paragraph of its own. This is especially useful for displaying a string with an element that has no text of its own but is only a parent for other paragraphs.

For example, suppose you want FrameMaker to display heads automatically for syntax descriptions that can have several paragraphs. If you have a `Syntax` element that is a parent for the paragraphs in a description, you can set up a prefix that provides a head appropriate for the context of `Syntax`:

Element (Container): Syntax
General rule: Para+
Prefix rule
1. **If context is:** Synopsis
Prefix: Synopsis and contents
Else, if context is: Args
Prefix: Arguments
Else, if context is: Examples
Prefix: Examples
2. **In all contexts.**
Text range.
Font properties
Weight: Bold

A syntax description begins with a bold head in a separate paragraph.

Format rules for first paragraph in element
1. **In all contexts**
Basic properties
Paragraph spacing
Space below: 4pt

The head paragraph has extra space below it.

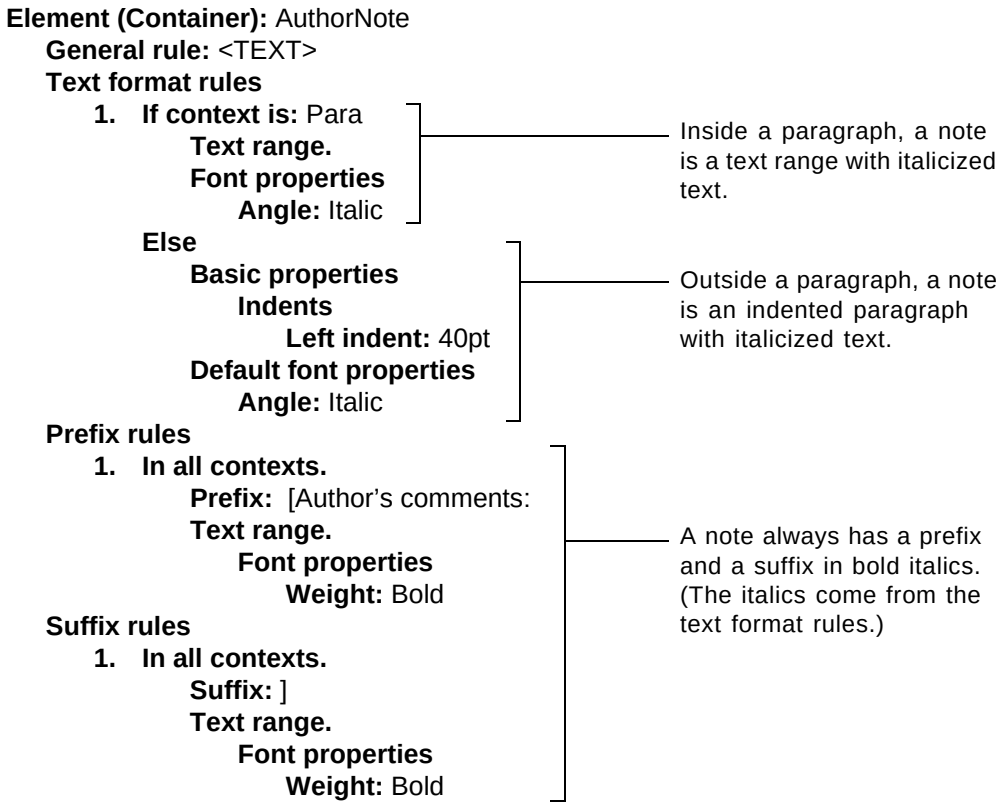
In each description, the boldfaced string *Synopsis and contents*, *Arguments*, or *Examples* appears in a paragraph by itself above the first child `Para`.

Note that in a prefix or suffix rule you can specify formatting changes only to font properties. If you are displaying a prefix or suffix in a paragraph by itself and want to apply paragraph changes to it such as space above or below, give the parent element a first format rule (for

a prefix) or a last format rule (for a suffix). The prefix is the first paragraph in the parent, or the suffix is the last paragraph in the parent. In the `Syntax` example above, the first format rule puts 4 points of space below the prefix head.

A prefix or suffix for a text range or a paragraph

A single prefix or suffix can work with both a text range and a paragraph. For example, suppose you want to allow author annotations that could be run in with regular text or set off in a paragraph of their own. You can define the element to be formatted as a text range or as a paragraph, and the prefix and suffix will apply in either case:



The prefix string in the example has a space at the beginning and the suffix string has a space at the end, so that within a paragraph the annotation will have extra space on each side.

Note that FrameMaker first applies the text format rules to the entire element and then applies the prefix and suffix rules. In the example above, the prefix and suffix strings are in bold italics because they get their angle property (italics) from the text format rules.

Attributes in a prefix or suffix rule

You can refer to an attribute in a prefix or suffix rule, and the value from the attribute is written as the text string. If the attribute has only a default value, that value is used. This allows you to define one prefix or suffix for different places in a document.

To use an attribute value, insert an `AttributeValue` element in `Prefix` or `Suffix`. The text <`$attribute[]`> appears in the prefix or suffix definition. Include one or more of the following building blocks in the definition:

<$attribute[*attrname*(:*elemtag1*, *elemtag2*, ...)]>

where *attrname* is the name of an attribute and each optional *elemtag* is the tag of an element that has the named attribute.

If you do not include an element tag, FrameMaker uses the value for the named attribute in the current instance of the element. For example, this definition displays the value of the `Security` attribute in the current element:

Prefix: <$attribute[Security]>

If you do include element tags, FrameMaker uses the value for the named attribute in the closest containing element with a tag that matches *elemtag* and an attribute that matches *attrname*. For example, this definition displays the value of the `Security` attribute in the closest containing `Item` or `LabelPara` that has the attribute:

Prefix: <$attribute[Security: Item, LabelPara]>

The element specification can include a context label in parentheses. FrameMaker finds the closest containing element with a tag that matches *elemtag* and the specified context label. For example, this definition displays the value of the `Security` attribute from the closest containing `Head` element that has a `Chapter Level` context label and a `Security` attribute:

Prefix: <$attribute[Security: Head(Chapter Level)]>

If you use empty parentheses or the token <`nolabel`> in parentheses, FrameMaker finds only elements without a context label. For example, the specification `Head()` or `Head(<nolabel>)` finds `Head` elements with no label.

When to use an autonumber, prefix or suffix, or first or last rule

You can display a fixed string of text with an element by using an autonumber or a prefix or suffix. An autonumber or prefix can be at the beginning of a paragraph; an autonumber or suffix can be at the end of a paragraph. But a prefix or suffix provides additional flexibility because you can also display the string inside a paragraph (with a text range element) or in a paragraph of its own (with a parent element). In addition, a paragraph can have both a prefix and a suffix but can have an autonumber only at one end.

A first or last format rule applies special paragraph formatting to the first or last paragraph in an element.

Here are a few cases in which you may want to use an autonumber, a prefix or suffix, or a first or last rule:

- To display a text string at the beginning or end of an element formatted as a paragraph, define the text as an autonumber or a prefix or suffix for the element. (For example, you might put *Important:* at the beginning of a paragraph.)

  If you want the string to appear with the first or last paragraph in a parent, define an autonumber in a first or last format rule for the parent. (For example, you might put *Important:* at the beginning of the first paragraph in a set-off note with several paragraphs.)

- To display a text string in the middle of an element formatted as a paragraph without breaking the paragraph, define the text as a prefix or suffix for a text range child element. (For example, you might put quotation marks around a quotation text range, or separate fields in a bibliography entry with appropriate punctuation and spaces.)

- To display a text string in a paragraph of its own at the beginning or end of a parent element that is only a container for paragraph elements, define the text as a prefix or suffix for the parent. (For example, you might put a head at the top of a section.)

  You can specify font changes for the string in the prefix or suffix rules. If you want to apply paragraph-formatting changes to the string's paragraph, define the changes in a first or last format rule for the parent.

For more information, see:

- "Numbering properties" on page 153 (for autonumber strings)
- "Writing first and last format rules" on page 156
- "Defining prefixes and suffixes" on page 158

Defining a format change list

You can describe a set of changes to paragraph format properties in a *format change list* and then refer to the list from an element definition. Since you need to describe the changes only once, this is helpful when two or more format rules use the same set of changes. A format change list can describe all the same properties that you can write out in a format rule clause.

To define a format change list, insert a `FormatChangeList` element at the same level as `Element` elements. For each group of properties you want to change, insert the element corresponding to the group (such as `PropertiesBasic`) and describe the changes. For a summary of the properties, see "Specifications for individual format properties" on page 146.

You may want to organize all the format change lists together in a separate section in the EDD. For information on using sections, see "Organizing and commenting an EDD" on page 90.

A format change list typically contains a general-purpose set of changes used by different kinds of elements. For example, this list changes the amount of spacing and indentation and can be applied to any kind of display text, such as examples and long quotations:

Format change list: DisplayText
- **Basic properties**
  - **Paragraph spacing**
    - **Space above:** 12pt
    - **Space below:** 12pt
  - **Indents**
    - **First indent position relative to left indent:** 0pt
    - **Move left indent by:** +12pt

You can refer to the format change list by name in text format rules, in first or last format rules, or in prefix or suffix rules for an element that is formatted as a paragraph or text range. For information on referring to a list, see "Defining the formatting changes in a rule" on page 143.

When you refer to a format change list in an element definition, only the changes that are appropriate for the current element apply. For example, the following list defines changes for code segments that can be formatted as either paragraphs or text ranges. If you refer to the list from a rule clause that formats the code as a text range, the font properties from the list apply but the basic properties do not:

Format change list: Code
- **Basic properties**
  - **Tab stops**
    - **Relative tab stop position:** +12pt
      - **Alignment:** Left
    - **Relative tab stop position:** +24pt
      - **Alignment:** Left
- **Default font properties**
  - **Family:** Courier
  - **Pair kerning:** No

If you apply this change list to a text range element, only the font properties are used.

Only the properties that are used from a change list are passed on to descendants.

When an end user imports element definitions, FrameMaker generates a Format Change List Catalog in the document from named change lists in the EDD. Each time the user re-imports definitions, a new catalog overwrites the existing one in the document. Importing formats does not affect the catalog. If the user removes the structure from a document, the Format Change List Catalog is also removed.

Some properties in format change lists have minimum and maximum limits. Font sizes, for example, must fall within the inclusive range 2 to 400 points. If you set a value that is outside an allowed range (either by specifying an absolute value or by calculating a new relative value), FrameMaker changes the value to the minimum or maximum. With some

values, you can also set your own limits. See "Setting minimum and maximum limits on properties," next.

Setting minimum and maximum limits on properties

When defining changes to individual properties in a format rule or in a format change list, you can use relative values for some of the properties in the Basic, Font, and Table Cell groups. A relative value is added to the property's current value to determine a new value. For example, if a `Head` element inherits a paragraph format with a font size of 12 points and you specify a `SizeChange` value of 2 points for the element, the new font size is 14 points.

If a relative value is applied to a property again and again, it is possible for the property's value to become unreasonably large or small. In the example of the `SizeChange` value of 2, the font size for heads becomes 20 points after four applications of the relative value—which may be larger than you have in mind for any section head.

You can set minimum and maximum limits on any properties that can use relative values in text format rules and change lists. A limit you set also applies if you specify an absolute value for the property.

To set minimum and maximum limits on properties, insert a `FormatChangeListLimits` element as the last child element in `ElementCatalog`. For each limit you want to set, insert an element for a set of limits (such as `LeftIndentLimits`) and then insert at least a `Minimum` or `Maximum` child element and type the limiting value. For example:

Limit values for format change list properties
- **Left indent**
  - **Minimum:** 1"
  - **Maximum:** 3"
- **First indent**
  - **Maximum:** 1.5"
- **Font size**
  - **Minimum:** 6 pt
  - **Maximum:** 20 pt
- **Line spacing**
  - **Minimum:** 12 pt

If a relative value in the EDD causes a property's value to be calculated beyond the limit you set, or if you enter an absolute value that is beyond the limit, the value is set to the specified minimum or maximum. For example, if you set a minimum font size of 6 points and an element's font is recalculated to 4 points, the font changes only to 6 points in the element.

Properties that can use relative values have a minimum and maximum limit already built in. You can further limit a built-in range, but you cannot extend it. These are the limits you can define and their built-in minimum and maximum values:

Set of limits	Minimum / maximum
`FirstIndentLimits`, `LeftIndentLimits`, or `RightIndentLimits`	0" / 39"
`LineSpacingLimits`, `SpaceAboveLimits`, or `SpaceBelowLimits`	-32,767" / 32,767"
`TabStopPositionLimits`	0" / 39"
`FontSizeLimits`	2 pts / 400 pts
`SpreadLimits` (Renamed `TrackingLimits`; still works for backward compatibility)	-1000% / 1000%
`TrackingLimits`	-1000% / 1000%
`SpreadLimits`	-1000% / 1000%
`CellMarginLimits`, `BottomCellMarginLimits`, `LeftCellMarginLimits`, `RightCellMarginLimits`, or `TopCellMarginLImits`	0 pt / 32,767 pts

The minimum and maximum limits are global and apply to a value wherever it occurs in the EDD. You cannot set limits for a value in a particular format rule or change list.

For a summary of the properties available in format rules and format change lists, see "Specifications for individual format properties" on page 146.

Debugging text format rules

After writing text format rules, you should try them out by importing the EDD into a sample document with text. If any text is not formatted the way you expect, check the EDD for these errors:

- Typing errors in the tags of paragraph formats, character formats, or format change lists in formatting specifications, or in the element tags or sibling indicators in context specifications
- Typing errors or incorrect child elements in specifications for individual format properties
- Duplicated context specifications
- Format rule clauses that are not in specific-to-general order

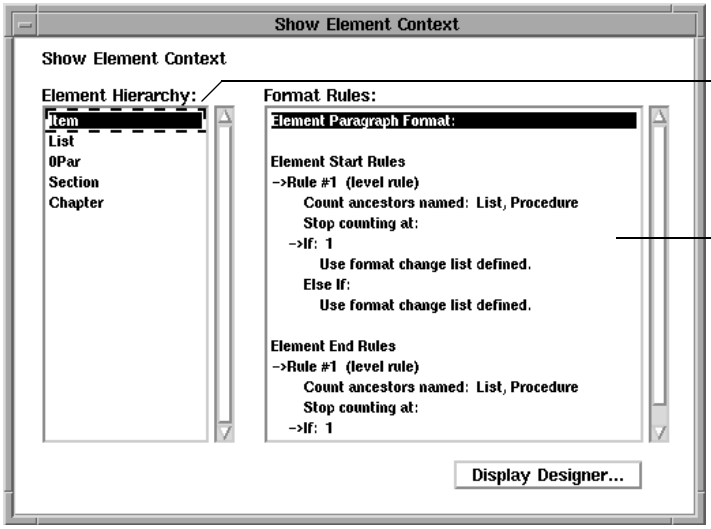
You might also need to look at formats in the document or the template to see if they are defined the way you expect. Check the definitions in these places:

- Paragraph formats in the Paragraph Designer (Format menu)
- Character formats in the Character Designer (Format menu)

If FrameMaker identifies any problems when you import an EDD, it produces a log file of warnings and errors. For information on how to work with this file, see "Log files for imported element definitions" on page 104.

As you examine the formatted contents of the document, you can use the Show Element Context dialog box to find out what format rules are being applied to text in a particular instance of an element. To open the dialog box, choose Show Element Context from the File>Developer Tools submenu.

Show Element Context lists the hierarchy of the current element on the left, beginning with the current element on top. If you select an element in the list, the right side of the dialog box shows that element's format rules. The tag of the element paragraph format in use appears on top, and below it are the format rules that modify the paragraph format. The rule clauses that apply to the particular element instance have an arrow pointing to them.



The scroll box on the right can show text format rules, first and last format rules, and prefix and suffix rules. If the selected element is an object, such as a table or cross-reference, the list shows object format rules instead. For an example of this dialog box with object rules, see "Debugging object format rules" on page 197.

To display the formatting properties for a paragraph format or character format, select the format in the list on the right and click Display Designer.

11 Attribute Definitions

Attributes provide a way to store descriptive information with an instance of an element that is not part of the element's content. In FrameMaker, you can define attributes for many purposes—such as to record the current status of an element's content, to allow cross-referencing between elements with ID attributes, or to specify how an element is to be formatted. You can define attributes for any element in FrameMaker.

Attribute definitions in FrameMaker translate to attribute declarations in markup. If you move documents between FrameMaker and XML or SGML and conform to the markup requirements, your attributes and their values are usually preserved. FrameMaker provides a default translation for all of its attribute types, and you can modify the translation by using read/write rules.

In this chapter

This chapter explains how to define attributes in FrameMaker. In the outline below, click a topic to go to its page.

Background on attributes in FrameMaker documents:

- "Some uses for attributes" on page 169
- "How an end user works with attributes" on page 170

Syntax of attribute definitions:

- "Writing attribute definitions for an element" on page 171

Attributes for special purposes:

- "Using UniqueID and IDReference attributes" on page 176
- "Using attributes to format elements" on page 180
- "Using attributes to provide a prefix or suffix" on page 182

Some uses for attributes

One of the most common uses for attributes is to record information about an instance of an element that an end user may want to inspect or update while editing the document. This information describes the element's content in some way. For example:

- A security attribute in a memo element can tell the level of classification for the memo's contents (`Security=Unclassified`).

- A status attribute in a section element can describe the current review stage of the section's contents (`Status=Alpha`).
- An author attribute in a chapter element can identify the author of the document (`Author=pjr`).
- A size attribute in a graphic element can specify the width of a frame (`ArtWidth=8.5`). For XML you provide a CDATA value, and for SGML you can provide a number value.

Attributes can store source and destination information for elements. These are often used for cross-referencing between elements. For example:

- An identifier attribute in a head element can uniquely identify the element as a source for cross-references (`ID=Intro`).
- A reference attribute in a cross-reference element can store the ID of the source element that is referred to (`Reference=Intro`).

You can also use attributes to determine the appearance of an element in a FrameMaker document. For example:

- A type attribute in a list element can specify whether the list should be numbered or bulleted (`Type=Bulleted`).
- A prefix attribute in a note element can provide a text string to display before the element's content (`Prefix=Important`).

How an end user works with attributes

In a document, an end user provides values for particular instances of attributes. The user can also validate the document to be sure that the attribute values meet the criteria of your definitions. To develop an environment that is reasonable for the user, it may help to understand a few things about how the user edits and validates attributes.

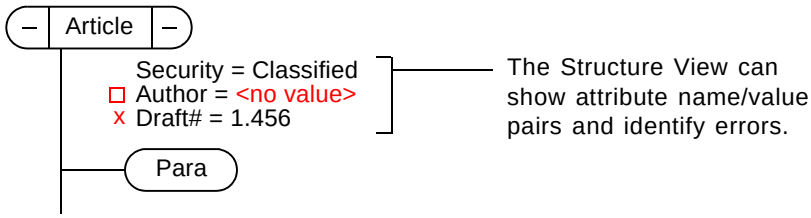
An end user supplies values for an attribute in the Attributes dialog box. The dialog box shows information from the attribute definition—name, type, range or choices, and default value—to guide the user to enter values that are appropriate. If the user tries to enter a value that is not valid according to the definition, FrameMaker does not accept the value and displays an alert.

You can also define read-only attributes with values that are set by an structure API client or by a FrameMaker cross-reference. An end user cannot modify read-only attributes.

Even though an end user cannot enter invalid attribute values in the Attributes dialog box, it is still possible to end up with inappropriate values or to have attributes in an element that is not defined to contain them. This usually happens because the user has pasted values or attributes or imported a new Element Catalog with different attribute definitions, or because of changes made to the document by a structure API client.

The end user can see attribute name/value pairs for each element in the Structure View. The view displays all attributes, no attributes, or only attributes with required and specified

values, whichever the user requests. Invalid values are identified by an *x* next to the attribute name. If an attribute is missing a required value, the view shows a hole next to the name. (The error information is in red on a color monitor.) For example:



FrameMaker also identifies errors involving attributes when the end user validates the document. For more information on the Structure View and how a user works with attributes, see the FrameMaker user's manual.

Writing attribute definitions for an element

To make attributes available for an element, you need to define the attributes as part of the element's definition. You can define attributes for any element in FrameMaker.

An element definition can have a list of attribute definitions. Within the list, each definition must have a name, a type, and specification of whether an attribute value is required or optional. If the attribute is of type `Choice`, the definition must also include a list of possible values. For example:

Element (Container): Article
General rule: Para+, Section*
Attribute list
1. **Name:** Author **String** **Required**
2. **Name:** Security **Choice** **Optional**
   Choices: Top Secret, Classified, Unclassified

Each attribute must have a name, a type, and a required or optional specification.

A Choice attribute also needs a list of values.

Element (Graphic): ImportedArt
Attribute list
1. **Name:** Width **Real** **Optional**
   Range: From 6 **to** 11
   Default: 8.5

You can optionally define a range of possible values (for the numeric attribute types) and a default value (if a value is optional). You can also make any attribute read-only.

To write attribute definitions for an element, insert an `AttributeList` element. (If the element is a container, table, table part, or footnote, the attribute list goes after the structure rules.) When you insert the `AttributeList` element, the first `Attribute` child element is inserted automatically. Define the first attribute, and then insert and define additional `Attribute` elements as necessary. The definitions are numbered automatically.

Attribute name

When you insert an `Attribute` element, the `Name` child element is inserted automatically along with it. Type the attribute name in the `Name` element. For example:

1. **Name:** Author

Give an attribute a name that is self-explanatory. Although different elements can have attributes with the same name, it's good practice to use the same name only for the same semantics. Attribute names are case-sensitive, and they cannot contain white-space characters or any of these special characters:

```
( ) & | , * + ? < > % [ ] = ! ; : { } "
```

An attribute name can have up to 255 characters in FrameMaker, but you should try to keep the names concise. End users often display attribute names with their elements in the Structure View.

SGML: If you plan to export documents to SGML, you may want to define attribute names that conform to the naming rules and the maximum name length permitted by the concrete syntax you'll be using in SGML. If you prefer names that do not adhere to the SGML conventions, you can provide read/write rules to convert them to SGML equivalents when you export. For more information, see "Naming elements and attributes" on page 231.

Attribute type

The attribute type determines what kind of values are allowed in the attribute. You can specify attribute types for string values, numeric values, and IDs and their references. Insert one of the type elements after the name in the attribute definition. For example:

1. **Name:** Author **String**

These are the attribute types available:

Type	Values allowed in the attribute
`String`	An arbitrary text string
`Strings`	One or more arbitrary text strings
`Integer`	A whole number, which may be signed (optionally restricted to a range of values); valid integers: `224`, `+795`, `-1024`
`Integers`	One or more integers (optionally restricted to a range of values)
`Real`	A real number (optionally restricted to a range of values), can use e notation to express 10 to the power of *x*; valid real numbers: `23`, `.23`, `0.23`, `2.3e-1`, `.023e+01`
`Reals`	One or more real numbers (optionally restricted to a range of values)
`Choice`	A value from a list of predefined choices

Type	Values allowed in the attribute
`UniqueID`	A string that uniquely identifies the element
`IDReference`	A string used elsewhere as the value of a `UniqueID` attribute
`IDReferences`	One or more references to `UniqueID` attributes

Note the following about these attribute types:

- With the `Choice` attribute type, you also need to provide a list of possible values in the attribute definition. With the numeric types, you have the option of constraining the values to a range. For details, see "List of values for Choice attributes" on page 175 or "Range of values for numeric attributes" on page 175.
- The numeric types allow only integer or real values and no additional strings. This allows for precise validation in FrameMaker, because the software can check that the values fall within a numeric range. If you want end users to have the flexibility to enter values such as `3 in` or `60 mm` and do not need to limit them to a range, use the `String` attribute type instead of a numeric type.
- The `UniqueID`, `IDReference`, and `IDReferences` types are typically used for element-based cross-referencing. For information on working with these types, see "Using UniqueID and IDReference attributes" on page 176.
- The multiple-token types `Strings`, `Integers`, and `Reals` are for attributes that need to store a set of information. For example, you can use a `Reals` attribute to store dimensions for an imported graphic object—an end user might enter two tokens that together define the height and width of the object.

In a document, if an end user tries to enter a value that is not allowed by the attribute type, FrameMaker does not accept the value and displays an alert describing the problem. If the user enters an invalid value by other means (such as pasting), FrameMaker identifies the attribute as invalid. (A duplicate `UniqueID` value is discarded.)

SGML: All attributes in FrameMaker translate to attributes in SGML. FrameMaker and SGML have different attribute types available however, and the types generally do not translate one to one. Multiple SGML attribute types can translate to the same FrameMaker type, and vice versa. For more information, see Chapter 16, "Translating Elements and Their Attributes."

Specification for a required or optional value

You need to specify whether or not an attribute requires a value for each instance of the element in a document. Insert a `Required` or `Optional` element after the type element in the attribute definition. For example:

1. Name: Author **String** **Required**

In a document, if an attribute requires a value but does not have one, FrameMaker identifies the attribute as invalid.

If a value is optional, you can assign a default value to the attribute. A default value can be used to control the element's formatting or to work with a structure API client. For more information, see "Default value" on page 176.

Hidden and Read-only attributes

You may sometimes want to restrict end users from editing the value of an attribute, and let FrameMaker or a structure API client provide the value instead. This is especially helpful for an attribute you want to use for precise tracking of particular element instances or changes in a document.

For example, FrameMaker can set an attribute value for elements used in cross-referencing. You specify read-only for a `UniqueID` attribute that stores an ID as source information for its element. Then when an end user inserts an element-based cross-reference to an instance of the element, FrameMaker sets the attribute's ID for that instance. For information on using an attribute in this manner, see "Using UniqueID and IDReference attributes" on page 176.

Your structure application might store information in attributes that is of no interest to the end user. In that case, you can make the attribute hidden. Not only is the end user kept from editing the attribute value, he or she will never see the attribute or its value.

These are possible uses for read-only or hidden attributes with values set by an API client:

- The attribute stores key information in a structured document that is a representation of a database. For example, a `UniqueID` attribute in FrameMaker might store an object ID with data extracted from a database. You may want to preserve the ID so that you can export the data back to its source location in the database.
- The attribute stores information about the version of a document. For example, some version-control schemes have an `IDReference` attribute pointing to each element in a document, and the attribute is updated for each revision.

To make an attribute read-only, insert a `ReadOnly` element after the `Required` or `Optional` element in the attribute definition. For example:

1. Name: Source **UniqueID Optional Read-only**

To make an attribute hidden, insert a `Hidden` element after the `Required` or `Optional` element in the attribute definition. For example:

1. Name: Source **UniqueID Optional Hidden**

> If you make an attribute read-only or hidden, we suggest that you specify the value to be optional. If a value is required and an instance of the attribute does not have a value, the document will show a validation error that the end user cannot correct.

Although a read-only or hidden attribute is not editable in the FrameMaker user interface, you can still edit its value with a structure API client. For information on API clients, see the *Structure Import/Export API Programmer's Guide*.

List of values for Choice attributes

For an attribute of type `Choice`, you need to provide a list of possible values for the end user. Insert a `Choices` element after the `Required` or `Optional` element (or after the `ReadOnly` element if the definition has one). Then type the possible values, separating them with a comma and a space. For example:

1. **Name:** Security **Choice** **Required**
   Choices: Top Security, Classified, Unclassified

The tokens can be strings of up to 255 characters. They can have white-space characters but cannot have any of these special characters:

```
( ) & | , * + ? < > % [ ] = ! ; : { } "
```

If you're using attribute values to format elements, the order of the values in the list may matter. When specifying attribute values in format rules, you can use greater-than or less-than operators to test the location of the current value in the `Choices` list. For more information, see "Attribute indicators" on page 136.

> **SGML:** If you plan to export documents to SGML, you may want to define values that conform to the naming rules and the maximum name length permitted by the concrete syntax you'll be using in SGML. If you prefer values that do not adhere to the SGML conventions, you can provide read/write rules to convert them to SGML equivalents when you export. For more information, see "Renaming attribute values" on page 235.

In a document, the values appear in the Attribute Value drop-down list in the Attributes dialog box when this attribute is selected. An end user chooses from the list to enter one of the values in the attribute. If the user enters an invalid value through other means (such as pasting), FrameMaker identifies the attribute as invalid.

Range of values for numeric attributes

If an attribute's type is `Integer`, `Integers`, `Real`, or `Reals`, you can optionally provide a range of possible values. Insert a `Range` element after the `Required` or `Optional` element (or after the `ReadOnly` element if the definition has one). A `From` child element is inserted automatically. Type the beginning value, insert a `To` element, and then type the ending value. For example:

1. **Name:** ArtWidth **Real** **Optional**
   Range: From 6 **to** 11

The values you enter must be appropriate for the type of attribute—that is, either all integers or all real numbers. The range is inclusive.

In a document, the Attributes dialog box shows the range of values when this attribute is selected. If the end user tries to enter a value outside the range, FrameMaker does not accept the value and displays an alert describing the problem. If the user enters an invalid

value by other means (such as pasting), FrameMaker identifies the attribute value as invalid.

Default value

If a value is optional in an attribute, you can specify a default value. FrameMaker can use the default value for formatting by attributes if an instance of the attribute does not have a value. Insert a `Default` element as the last child element in the attribute definition, and then type the value. For example:

1. **Name:** ArtWidth **Real** **Optional**
   Range: From 6 **to** 11
   Default: 8.5

If the attribute is one of the multiple-token types (`Strings`, `Integers`, `Reals`, or `IDReferences`), you can specify more than one value token for the default value. For each additional token, insert a `Default` element and type the value. When you add a second token, the `Default` label in the EDD changes to plural. For example:

1. **Name:** ArtWidth **Real** **Optional**
   Range: From 6 **to** 11
   Default: 8.5
   11

For information on how FrameMaker can use attribute values in formatting, see "Using attributes to format elements" on page 180. For information on structure API clients, see the *Structure Import/Export API Programmer's Guide*.

Using UniqueID and IDReference attributes

A `UniqueID` attribute stores a string that uniquely identifies an instance of an element in a document or book. In FrameMaker, this ID value is often used as a source address for cross-references to the element. A structure API client or a structure application might also use the ID for other purposes.

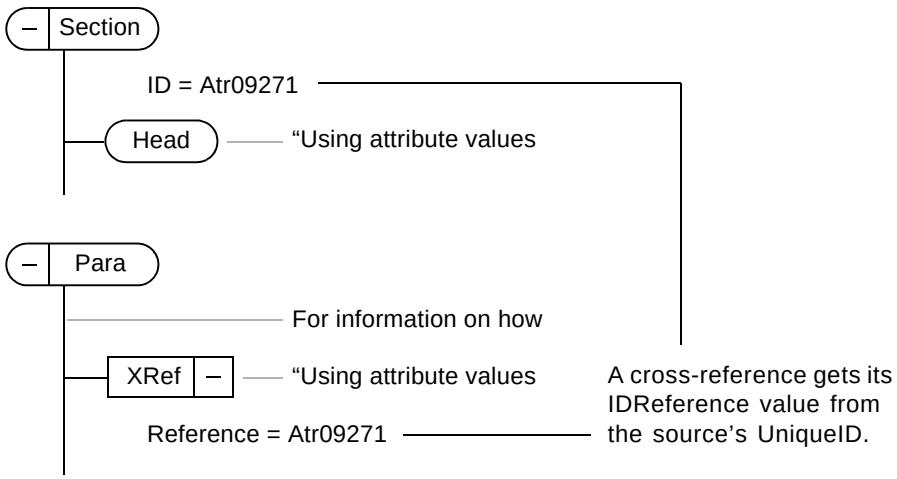
When a `UniqueID` attribute is used for element-based cross-referencing, the elements that reference the ID have an `IDReference` attribute. The `IDReference` attributes store the address from the source's `UniqueID` attribute.

FrameMaker provides a mechanism for automated cross-referencing using the `UniqueID` and `IDReference` attributes. You need to assign a `UniqueID` attribute to elements that will likely be the sources of references (such as chapters, sections, and tables) and an `IDReference` attribute to the cross-reference object elements that refer to the sources. When an end user inserts a cross-reference to a source, FrameMaker automatically fills in the cross-reference's `IDReference` attribute with the value from the source's `UniqueID` attribute.

With FrameMaker cross-referencing, information from the source appears in the cross-reference in the document. For example, a cross-reference might show the heading and the

beginning page number of a section it refers to. To determine what information from a source appears in a cross-reference, you define a *cross-reference format*. For more on these formats, see "Setting a cross-reference format" on page 194.

This example shows how a FrameMaker element-based cross-reference appears in the Structure View:



If an end user inserts a cross-reference to a `UniqueID` that does not yet have a value, FrameMaker generates an ID value for the `UniqueID` and the `IDReference`.

Attributes for cross-referencing can help end users keep track of their references and what they point to. Because the `UniqueID` and `IDReference` attributes on both ends of a cross-reference store the same value, a user can look at a cross-reference element in the Structure View and see information about the reference's source. In addition, if the user knows the `UniqueID` value of a source, he or she can find all cross-references to the source by searching for other elements with that value in their `IDReference` attribute.

You may also want to make it possible for end users to work with `IDReference` attributes as informal pointers to elements with a `UniqueID`. In this case, the pointers are not FrameMaker cross-references, so information from the source (such as heading and page number) does not appear in the document—but the user can still look at the Structure View to see pointers to related information. For example, suppose you define an `IDReference` for a `Para` container element. To keep track of information from another section in an instance of the `Para`, a user can manually enter the section's ID in the `IDReference` for

Para (using the Attributes dialog box). He or she can go to the source from the Para later by searching for the element with the ID.

SGML: When importing and exporting between FrameMaker and SGML, the UniqueID and IDReference attributes are preserved. UniqueID attributes in FrameMaker translate to ID attributes in SGML, and IDReference attributes translate to IDREF attributes. For more information, see Chapter 20, "Translating Cross-References."

UniqueID attributes

You can assign a UniqueID attribute to any element in FrameMaker. If you plan to use the element as a source for cross-references, the element will likely be a chapter, section, table, or figure. An element can have only one UniqueID attribute.

In the attribute definition, insert a UniqueID element after the name. For example:

Element (Container): Section
General rule: <TEXT>
Attribute list
1. Name: ID **UniqueID** **Optional** **Read-only**

The value in an instance of a UniqueID attribute must be unique for this attribute type in a document or book. Even if a document has two different elements (such as Section and Chapter) that have a UniqueID attribute, you cannot have any duplication of ID values.

It is possible for a document to end up with IDs that are not unique—for example, if the end user shows hidden text that contains an element with a conflicting ID, or if you change an attribute type to UniqueID and instances of the attribute already have duplicate values. FrameMaker identifies duplicate IDs as invalid.

An end user can provide ID values, or FrameMaker can generate the values. Each method has its advantages:

- If a user provides the IDs, he or she can use values that are meaningful (whereas an ID that FrameMaker generates is a random string). This can make it much easier to remember IDs and to recognize them in the Cross-Reference dialog box and the Structure View.
- If FrameMaker generates the IDs, the IDs are virtually guaranteed to be unique within a document or book, and they will remain unique because a user cannot edit them. The IDs that FrameMaker generates also conform to the SGML reference concrete syntax.

Although UniqueID attributes are often used in sources for cross-references, an element with this attribute is not required to have a reference to it.

When an end user provides an ID

An end user can provide an ID value of up to 255 characters in the Attributes dialog box or in some cases by pasting (unless the attribute is read-only). FrameMaker tries to ensure

the uniqueness of IDs as the user edits a document. For example, if the user enters an ID that is not unique, FrameMaker does not accept the value and displays an alert. If the user pastes an element with an ID that is not unique, the pasted element loses its attribute value.

FrameMaker cannot test for whether an entered ID is used in a different document in a book, especially since one document can be in more than one book. When the user validates a book, however, FrameMaker reports conflicts between IDs across documents in the book.

An end user can edit an existing ID in the Attributes dialog box (unless the attribute is read-only). Any cross-references already pointing to that element may become unresolved. FrameMaker accepts the value if it is unique, but warns the user about possible unresolved references. The user can check for unresolved cross-references by searching for an element with an `IDReference` attribute value equal to that of the replaced ID.

> **SGML:** If you plan to export documents to SGML, the SGML naming rules will likely allow fewer characters—and different characters—for attribute values. Refer to the concrete syntax you'll be using in SGML for the maximum name length and the characters allowed. To ensure that your end users conform to the concrete syntax, you may want to prepare recommendations on entering values. Remind users to begin IDs with a name-start character and to use only name characters thereafter.

When FrameMaker generates an ID

If an end user inserts a cross-reference element and the source's `UniqueID` attribute does not yet have a value, FrameMaker provides a unique value for it. The value is entered in both the `UniqueID` attribute and the `IDReference` attribute pointing to it.

An ID that FrameMaker generates is an eight-character string that begins with a capital letter and then has capital letters and digits. If you are exporting documents to markup, these IDs conform to the SGML reference concrete syntax.

A generated ID is unique in its document. The ID has the form *pppxxxxx*, where *ppp* is derived from the name of the document and *xxxxx* is a random string. Because two documents in a book must have different filenames, generated IDs in the documents will not conflict. If an end user renames a document, the *ppp* is recalculated for values that FrameMaker generates thereafter, but any existing values are not replaced.

FrameMaker automatically generates an ID value if the user inserts a cross-reference to the element, but does not insert an ID value. If you only want FrameMaker to have the ability to enter ID values, make the attribute read-only, thereby preventing the user from setting the attribute. For more information, see "Hidden and Read-only attributes" on page 174.

IDReference attributes

You can assign an `IDReference` or `IDReferences` attribute to any element in FrameMaker. If you plan to use the element as a FrameMaker cross-reference, the element should be a cross-reference object element.

In the attribute definition, insert an IDReference or IDReferences element after the Name element. For example:

Element (CrossReference): XRef
Attribute list
1. Name: Reference **IDReference** **Required**

In a document, when an end user inserts a cross-reference element with an IDReference and points the reference to an element with a UniqueID, FrameMaker automatically fills in the IDReference with the value from the source's UniqueID.

FrameMaker also allows an end user to point a cross-reference element to a paragraph source rather than to an element source. In this case, the value of the IDReference remains unspecified.

If you want to require end users to base all cross-references on elements rather than on paragraphs, provide an IDReference attribute for every cross-reference element, and make the attribute required. If a cross-reference points to a paragraph, its IDReference attribute does not have a value, and FrameMaker identifies the attribute as invalid.

SGML: For element-based cross-referencing in FrameMaker, you use the singular IDReference attribute, and one value is stored to describe the location of the source. Some SGML applications may require the IDReferences list attribute to store multiple values that build composite source information. Use IDReferences if you plan to share documents with one of these applications.

If you export an IDReferences list attribute to SGML, multiple-value source data for the attribute is preserved. If you import a multiple-value source from SGML, FrameMaker preserves the source data but uses only the first value.

You can define more than one IDReference attribute for an element. For example, you may want to do this to allow end users to work with the attributes as informal pointers to several sources. For a discussion of pointers that are not FrameMaker cross-references, see page 177.

It is possible for a document to end up with an IDReference value that does not match a UniqueID value in the document or book (usually because the end user has pasted the IDReference value or deleted the UniqueID value). FrameMaker identifies the IDReference as invalid.

Using attributes to format elements

You can refer to an attribute name/value pair in a format rule to identify instances of an element in a document. If an instance of the element has the attribute name and value specified, FrameMaker applies the format rule to it.

For example, suppose you want to be able to format the items in a List element two different ways—with bullets or with numbers. By giving the element an attribute that describes the type of List, you can allow an end user to specify the bulleted or numbered

type for each instance of the element and have the instance formatted according to that information. (If you did not use an attribute, you would need to define two separate `List` elements.)

In the `List` element definition give the element a `Type` attribute, and in the `Item` definition refer to the possible values for the attribute in the text format rules. The `Item` definition in this example specifies that each `Item` begins with a bullet if it appears in a `List` that has a `Type` attribute with the value `Bulleted`, or the `Item` begins with an incrementing number if it appears in a `List` that has a `Type` attribute with the value `Numbered`:

Element (Container): List
General rule: Item+
Attribute list
1. Name: Type **Choice** **Required**
Choices: Bulleted, Numbered

Element (Container): Item
General rule: <TEXT>
Text format rules
1. If context is: List [Type = "Bulleted"]
Numbering properties
Autonumber format: \b\t
Character format: bulletsymbol
Else, if context is: List [Type = "Numbered"]
1.1 If context is: {first}
Numbering properties
Autonumber format: <n=1>\t
Else
Numbering properties
Autonumber format: <n+>\t

The value of the attribute (from the List element) determines a context for text formatting.

You can refer to attribute values in object format rules as well as in text format rules. In this example, a `Table` uses `Format A` if its `Type` attribute has the value `Summary` or it uses `Format B` if its `Type` attribute has the value `Examples`:

Element (Table): Table
General rule: Title, Heading, Body
Attribute list
1. Name: Type **Choice** **Optional**
Choices: Summary, Examples
Default: Summary
Initial table format
1. If context is: [Type = "Summary"]
Table format: Format A
Else, if context is: [Type = "Examples"]
Table format: Format B

The value of the attribute (from the current element) determines a context for object formatting.

FrameMaker can use the default value of an attribute to determine context. In the `Table` example above, if `Type` is not specified for a table in a document, the table's initial format will be `Format A` (the format for the default value `Summary`).

Note that in both the `Item` and the `Table` examples, you could use `Else` rather than `ElseIf` to describe the second context. In the `Item` definition `Numbered` is the only other possible value for the `Type` attribute from `List`, and in the `Table` definition `Examples` is the only other possible value. When you use the `Else` specification, the label in the definition is simply `Else` and you do not specify a context such as `Type="Numbered"`.

You can test the values of multiple attributes by joining the specifications with an ampersand (&). For example, this specification is true if the element has a `Type` attribute with the value `Numbered` *and* a `Content` attribute with the value `Procedure`:

List [Type = "Numbered" & Content = "Procedure"]

In addition to testing for equality with attribute values, you can also use `!=` to test for inequality (with all attribute types) or a greater-than sign (>) or less-than sign (<) to test for comparison (with the `Choice` and numeric types).

If you use a greater-than sign or a less-than sign with a `Choice` attribute in a format rule, FrameMaker evaluates the name/value pair using the order in the list of values in the attribute's definition, with the "lowest value" being the one on the left. For example, this pair specifies any `Security` value that is to the left of `Classified` in the defined `Choices` list for the `Security` attribute:

Report [Security < "Classified"]

For more detailed information on attribute name/value pairs in context specifications, and the operators you can use with them, see "Attribute indicators" on page 136.

Using attributes to provide a prefix or suffix

A *prefix* is a text range that appears at the beginning of an element (before the element's content); a *suffix* is a text range that appears at the end of an element (after the content). An attribute value can provide the text for the prefix or suffix of a container. In this example, the `Note` definition specifies that the prefix is the current value of the element's `Label` attribute:

Element (Container): Note
General rule: <TEXT>
Attribute list
1. **Name:** Label **Choice** **Required**
Choices: Important, Note, Tip
Prefix rule
1. **In all contexts.**
Prefix: <attribute[Label]> ——— The value of the attribute provides a text string for the prefix.
Font properties
Weight: Bold

A reference to an attribute value can also include element tags. In this case, FrameMaker uses the value for the named attribute in the closest containing element with the specified tag. For example, this definition displays the value of the `Label` attribute in the closest containing `List` or `Section` that has the attribute:

Prefix: <attribute[Label: List, Section]>

If you list element tags and want to search in the current element, you need to include that element in the list of tags.

For more detailed information on attribute names in prefix and suffix definitions, see "Attributes in a prefix or suffix rule" on page 162.

12 Object Format Rules

A FrameMaker document uses special elements for tables, graphics, markers, cross-references, equations, and system variables. Each of these objects can have a formatting property in a document, such as a table format or an equation size. You can define this property in an object format rule for the element, and the format is applied automatically when an end user inserts an instance of the element in a document.

FrameMaker object format rules have no direct counterparts in markup. If you import a DTD, you can add format rules to the resulting EDD for the objects you plan to use in FrameMaker. If you import a markup document with a CALS `table` element that has a single `tgroup` element, and if the EDD in use has a definition for an element named `Table`, any table format specified in the EDD's `Table` definition is applied to the imported table.

If you export a document or EDD to XML or SGML, the object formatting information is not preserved. However, if you export to XML the software can generate a CSS file that captures the overall formatting of your FrameMaker document. This includes the paragraph formats used in table cells, for example. You specify whether or not to generate the CSS in the `structapps.fm` file. For more information, see "Managing CSS generation" on page 62.

In this chapter

This chapter explains how to write object format rules for tables, graphics, markers, cross-references, equations, and system variables in FrameMaker. In the outline below, click a topic to go to its page.

Background on object format rules:

- "Overview of object format rules" on page 186
- "Context specifications for object format rules" on page 187

Syntax and uses for object format rules:

- "Setting a table format" on page 190
- "Specifying a graphic content type" on page 191
- "Setting a marker type" on page 192
- "Setting a cross-reference format" on page 194
- "Setting an equation size" on page 194

- “Specifying a system variable” on page 195

Information to help you correct errors in format rules:

- “Debugging object format rules” on page 197

Overview of object format rules

An object element can have one format rule that specifies a formatting property for the element in a document. With the exception of system variables, these properties are not binding—they are initial suggestions for the end user. The rules apply only to new objects in a document and do not affect existing objects.

Unlike text format rules, an object format rule defines the property only for the current instance of an element and is not passed on through a hierarchy to other elements. Although an object can have only one format rule, the rule can have separate clauses that allow for context-specific variations. For example:

Element (Marker): GlossaryTerm
Initial marker type
1. **In all contexts.**
Use marker type: Glossary

Element (CrossReference): CrossRef
Initial cross-reference format
1. **If context is:** * < Table
Use cross-reference format: Page
Else
Use cross-reference format: Heading & Page

These are the formatting properties you can define for FrameMaker objects:

- A table format for a new table element. The format determines properties such as indentation and alignment, margins and shading, and ruling between columns and rows.
- A content type for a new graphic element. The content type specifies that the element is either an anchored frame or an imported graphic object. When an end user inserts the element, the dialog box that appears is either Anchored Frame or Import File.
- A marker type for a new marker element. The marker type specifies the purpose of the marker; some possible marker types are index, glossary, and hypertext.
- A cross-reference format for a new cross-reference element. The format determines the wording and punctuation for a reference, such as *See page 17*.
- An equation size for a new equation element. The equation size (small, medium, or large) controls the font size and horizontal spread of characters.
- A variable for a new system variable element. The variable can be one of the built-in system variables for dates and times, filenames, page and table information, and text for running headers and footers.

The end user can change a marker type, cross-reference format, or other property (except for a system variable) at any time, and the change is not considered to be a format rule override. If the user re-imports element definitions and turns on “Remove Format Rule Overrides,” the properties remain as the user has set them. An end user cannot change the variable in a system variable element.

Context specifications for object format rules

An object format rule can apply to all contexts in which the element occurs, or it can define particular contexts or the number of levels deep the element is nested in an ancestor. If the rule defines contexts or levels, it can have separate if, else/if, and else clauses for different possibilities. Each “in all contexts” rule and each if, else/if, and else clause specifies a formatting property.

For example:

Element (Table): Table
General rule: Title, Heading?, Body
Initial table format
1. **If context is:** * < Chapter
Table format: StandardTable
Else, if context is: * < Appendix
Table format: SyntaxTable

A format rule clause has a context specification...
...and a formatting property for that context.

A format rule or clause can have another format rule nested inside it, and can also include a context label to help end users select elements when inserting cross-references or preparing a generated list.

In most respects, the context specifications for object format rules are the same as they are for text format rules. This section describes the rules for “in all contexts” and the clauses for particular contexts, which are the specifications you’re most likely to use in object formatting. For information on nested rules, context labels, and count statements in level rules, see Chapter 10, “Text Format Rules for Containers, Tables, and Footnotes.”

All-contexts rules

A format rule can specify a format that applies to an element in all contexts in which it can occur. To write an all-context format rule, insert an `AllContextsRule` element, and then define the formatting changes for the rule.

In this example, an `InLineEquation` element uses the `Small` equation size no matter where the element occurs in a document:

Element (Equation): InLineEquation
Initial equation size
1. **In all contexts.**
Use equation size: Small

Context-specific rules

A format rule can define one or more possible contexts, with a format for each context. The contexts are expressed in separate if, else/if, and else clauses. When applying a format rule to an element, FrameMaker uses the first clause in the rule that is true for the instance of the element.

To write a context-specific format rule, insert a `ContextRule` element. An `If` element and a nested `Specification` element are inserted automatically along with it. (The `Specification` element does not have a label in the document window.) Type one or more element tags to define the `If` context, and then define the formatting changes for that context. If you need additional clauses in the format rule, you can insert and define any number of `ElseIf` elements, ending with one `Else` element.

Defining a context

When defining a context, you can name the parent element or a list of ancestors. For a list of ancestors, begin with the parent and then name successively higher-level ancestors, separating the element tags with less-than signs (<).

In this example, a `Filename` system variable element displays the full pathname of the current file if the element occurs in an `Item` in a `List` in a `Preface`, or it displays only the filename if it occurs anywhere else:

Element (System Variable): Filename
System variable format rule
1. **If context is:** Item < List < Preface
Use system variable: Filename (Long)
Else
Use system variable: Filename (Short)

The ancestors in a list can also be instances of the same element, to describe nesting within that element. For example, this specification is true if an element's parent is a `Section` that is a child element of another `Section`:

Section < Section

Note that a nesting specification of this type is true whenever the current element is nested in *at least* as many levels as shown in the rule. That is, `Section < Section` applies a formatting change if the current element is nested within two or more `Section` elements.

Wildcards for ancestors

Use an asterisk (*) as a wildcard to represent an unspecified number of successive ancestors in the hierarchy. For example, this specification is true if an element's parent is a `Section` and any one of the parent's ancestors is also a `Section`:

Section < * < Section

OR indicators

Use OR indicators (|) to test the specification for any ancestor in a group. Separate the element tags of the ancestors with an OR indicator, and enclose the group in parentheses.

For example, this specification is true if an element appears in a `List` within a `Preface` or a `Chapter`:

List < (Preface | Chapter)

Sibling indicators

Use a sibling indicator to describe an element's location relative to its siblings. You can use the indicator to describe the relationship of the current element to its siblings or of an ancestor element to its siblings. Enclose the sibling indicator in braces ({ }).

To describe the relationship of the current element to its siblings, type the sibling indicator and a less-than sign, and then continue with the parent and other ancestors. For example, this specification is true if an element is the first element in its parent `NumberList`:

{first} < NumberList

To describe the relationship of an ancestor to its siblings, append the sibling indicator to the ancestor's tag. For example, this specification is true if an element's parent is a `Section` in a `Chapter` and the `Section` immediately follows a `Title`:

Section {after Title} < Chapter

These are the sibling indicators you can use:

Indicator	Specification is true if the element is
`{first}`	The first element in its parent
`{middle}`	Neither the first element nor the last element in its parent
`{last}`	The last element in its parent
`{notfirst}`	Not the first element in its parent
`{notlast}`	Not the last element in its parent
`{only}`	The only element in its parent
`{before` *`sibling`*`}`	Followed by the named element or text content
`{after` *`sibling`*`}`	Preceded by the named element or text content
`{between` *`sibling1`*`,` *`sibling2`*`}`	Between named elements or text content
`{any}`	Anywhere in its parent (equivalent to no indicator)

The *`sibling`* argument with the `before`, `after`, and `between` indicators can be an element tag or the keyword <TEXT>. If you use <TEXT>, FrameMaker looks to see if the element is preceded or followed by text rather than by a sibling element. A string generated by a prefix, suffix, or autonumber is not considered to be text.

Attribute indicators

You can also use an attribute name/value pair in an object format rule clause to more narrowly define context. For the context specification to be true, an instance of the element

must have the attribute name and value specified. (If the element does not have an attribute value but the attribute is defined to have a default value, the default value is used.)

When an end user inserts an object element that has an attribute in its format rule, the Edit Attributes dialog box appears right away so that the user can provide an attribute value (if the user has the element set to display this dialog box automatically.).

To test an attribute of the current element, type the attribute name and value in brackets as the context specification. To use an attribute with an ancestor, type the attribute name and value in brackets after the ancestor tag. Separate the attribute name and value with an equal sign, and enclose the value in double quotation marks.

In the following example, a `Table` uses `Format A` if its `Type` attribute has the value `Summary`, or it uses `Format B` if its `Type` attribute has the value `Examples`:

Element (Table): Table
General rule: Title, Heading, Body
Attribute list
1. Name: Type **Choice** **Optional**
Choices: Summary, Examples
Default: Summary
Initial table format
1. If context is: [Type = "Summary"]
Table format: Format A
Else, if context is: [Type = "Examples"]
Table format: Format B

The value of the attribute determines a context for object formatting.

The syntax for using attributes in object format rules is the same as it is in text format rules. For details, see “Attribute indicators” on page 136.

Order of context clauses

When a context-specific format rule has more than one clause, keep in mind that FrameMaker applies the first clause in the rule that is true for the instance of the element. You must write rule clauses from the most specific to the most general.

For example, suppose you want to apply a format to a cross-reference when it appears in an `Item` inside a nested `List` element (a `List` inside a `List`). If you put the context specifications for the cross-reference in the following order, FrameMaker would never apply the second clause because an `Item` in a nested `List` also matches the first specification:

Item < List
Item < List < List

You get the effect you want by reversing the clauses.

Setting a table format

A FrameMaker table uses a table format to determine the basic appearance of the table—such as indentation and alignment, margins and shading in cells, and ruling between

columns and rows. Like paragraph formats and character formats, table formats are defined and stored in catalogs in the documents.

You can set an initial table format for new instances of a table element. When an end user inserts the table, the format is preselected in the Insert Table dialog box.

To set a table format, insert an `InitialTableFormat` element after the table element's structure rules (and after any attribute definitions). Then insert and define context elements as necessary, and for each context, insert a `TableFormat` child element and type the tag of a format. The tag must refer to a format stored in the Table Catalog of the documents. For example:

Element (Table): Table
General rule: Heading?, Body, Footing?
Initial table format
1. **If context is:** * < Examples
   Table format: TableWithLines
   Else, if context is: Item < List
   Table format: IndentedTable
   Else
   Table format: StandardTable

If you do not set a table format, the element uses `Format A`.

Make sure that a table format you use is consistent with the general rule for the table. In the example above, the table formats should not have a table title because the general rule does not allow one.

The initial table format is only a suggestion for the end user. The user can change a table to another format stored in the document (using either Insert Table or the Table Designer), and the change is not regarded as a format rule override. If the user re-imports element definitions with "Remove Format Rule Overrides" on, the table does not return to the format suggested by the EDD.

When importing markup documents, the table format may be specified through a mapping of an attribute to the fm property *table format*. See "is fm property" on page 424 for more information on this property. If this has been done, then the Initial table format rule in the EDD will be ignored. If no attribute in markup has been mapped to the fm property *table format*, then the Initial table format rule in the EDD will be applied.

Tables can also have text format rules that define formatting properties for text in descendant title and cell elements. For information on these format rules, see Chapter 10, "Text Format Rules for Containers, Tables, and Footnotes."

Specifying a graphic content type

A graphic element can be an anchored frame (which an end user can fill with any graphic object) or an imported graphic file. When a user inserts the element, the Anchored Frame dialog box or the Import File dialog box opens so that he or she can provide information

about the object. You can specify a content type for a graphic element, to determine which dialog box to open for new instances of the element.

The Anchored Frame dialog box sets the size, position, and alignment of the frame and other options such as cropping and floating. The Import File dialog box sets the name and location of the graphic file and specifies whether to import by reference or by copying.

To specify a graphic content type, insert an `InitialObjectFormat` element in the graphic element's definition (after any attribute definitions). Then insert and define context elements as necessary; and for each context, insert an `AnchoredFrame` or `ImportedGraphicFile` child element. For example:

Element (Graphic): Figure
Initial graphic element format
1. **If context is:** Item < Procedure
   Insert imported graphic file.
   Else
   Insert anchored frame.

You may want to define separate elements for imported graphics and anchored frames, and let the end user select the one that suits his or her purpose. In this case, use descriptive element tags as a guide for the user:

Element (Graphic): ImportGraphic
Initial graphic element format
1. **In all contexts.**
   Insert imported graphic file.

Element (Graphic): AnchFrame
Initial graphic element format
1. **In all contexts.**
   Insert anchored frame.

If you do not specify a graphic content type, the element uses the anchored frame type.

The initial content type is only a suggestion for the end user. The user can change an anchored frame to an imported file or vice versa (using either the Anchored Frame or the Import FIle dialog box), and the change is not regarded as a format rule override. If the user re-imports element definitions with "Remove Format Rule Overrides" on, the graphic does not return to the content type suggested by the EDD.

Setting a marker type

FrameMaker markers identify specific locations in a document—for example, to note sources for cross-references, indexes, and other generated lists or to identify active areas for hypertext commands. A marker's type signifies the purpose of the marker, such as cross-reference, index entry, or hypertext location.

You can set an initial marker type for a new instance of a marker element. When an end user inserts the element in a document, the specified marker type is preselected in the Insert Marker dialog box.

To set a marker type, insert an `InitialObjectFormat` element in the marker element's definition (after any attribute definitions). Then insert and define context elements as necessary, and for each context insert a `MarkerType` child element and enter the name of a marker type. For example:

Element (Marker): IndexEntry
Initial marker type
1. **In all contexts.**
Use marker type: Index

These are the marker types available:

Type	Purpose
`Author`, `Equation`, `Glossary`, or `Subject`	Identifies the source of an entry for a generated list or special type of index.
`Comment`	Attaches a nonprinting comment to a location.
`ConditionalText`	Identifies content that is conditional.
`CrossRef`	Marks a source location for a spot cross-reference. (Used only for a cross-reference source that is not an instance of an element or paragraph format.)
`HeaderFooter$1` or `HeaderFooter$2`	Marks a point in the flow where an end user wants the text of a header or footer to change. In a Running H/F system variable, the `$marker1` or `$marker2` building block reads this marker.
`Hypertext`	Attaches a hypertext command to a location.
`Index`	Identifies the source of an entry for a standard index.
`Type 11`	Identifies markup entities and processing instructions.
`Type 12`, ... `Type 25`	Identifies the source of an entry for an unspecified type of index or generated list, or identifies a source used by an structure API client.

A marker's text provides the content of the marker, such as the text for an entry in a generated list or the command syntax for a hypertext command. When the end user inserts an instance of the marker element, he or she enters the marker text.

If you do not set a marker type, the element is preset to whatever marker type the end user last selected in Insert Marker or the Marker window.

The initial marker type is only a suggestion for the end user. The user can change a marker to another type (using either Insert Marker or the Marker window), and the change is not

regarded as a format rule override. If the user re-imports element definitions with “Remove Format Rule Overrides” on, the marker does not return to the type suggested by the EDD.

Setting a cross-reference format

A cross-reference format determines the wording and punctuation for a cross-reference. For example, a format called `Heading & Page` might display a reference as *See "Defining a prefix or suffix" on page 17*. A format called `Page` might display the same reference simply as *See page 17*. Cross-reference formats are defined and stored in documents.

You can set an initial cross-reference format for a new instance of a cross-reference element. When an end user inserts the element in a document, the specified format is preselected in the Insert Cross-Reference dialog box.

To set a cross-reference format, insert an `InitialObjectFormat` element in the cross-reference element’s definition (after any attribute definitions). Then insert and define context elements as necessary, and for each context insert a `CrossReferenceFormat` child element and type the name of a format. The name must refer to a cross-reference format stored in the documents. For example:

Element (CrossReference): CrossRef
Attribute list
1. **Name:** Reference **IDReference** **Required**

Initial cross-reference format
1. **In all contexts.**
Use cross-reference format: Heading & Page

If you do not set a cross-reference format, the element is preset to whatever format the end user last selected in the Cross-Reference dialog box.

The initial cross-reference format is only a suggestion for the end user. The user can change a cross-reference to another format (using the Cross-Reference dialog box), and the change is not regarded as a format rule override. If the user re-imports element definitions with “Remove Format Rule Overrides” on, the cross-reference does not return to the format suggested by the EDD.

When importing markup documents, the cross-reference format may be specified through a mapping of an attribute to the `fm property` *cross-reference format*. See “is fm property” on page 424 for more information on this property. If this has been done, then the initial cross-reference format rule in the EDD will be ignored. If no attribute in the markup has been mapped to the `fm property` *cross-reference format*, then the Initial cross-reference format rule in the EDD will be applied.

Setting an equation size

FrameMaker provides three sizes for equations: small, medium, and large. The equation size controls the font size of expressions in the equation, the size of the integral and sigma

symbols, and the horizontal spread between characters. The properties for the sizes are set in the Equation Sizes dialog box in a document.

You can set an initial equation size for a new instance of an equation element. When an end user inserts the element in a document using the Element Catalog, the equation automatically has the specified size. (If a user inserts the element using the Small, Medium, or Large Equation command in the Equations palette, the command determines the equation size and can override the format rule.)

To set an equation size, insert an `InitialObjectFormat` element in the equation element's definition (after any attribute definitions). Then insert and define context elements as necessary, and for each context insert a `Small`, `Medium`, or `Large` child element. For example:

Element (Equation): DisplayEquation
Initial equation size
1. **In all contexts.**
Use equation size: Large

If you do not set an equation size, the element uses the size `Medium`.

The initial equation size is only a suggestion for the end user. The user can change an equation to another size (using the Object Properties dialog box), and the change is not regarded as a format rule override. If the user re-imports element definitions with "Remove Format Rule Overrides" on, the equation does not return to the size suggested by the EDD.

Specifying a system variable

System variables derive information such as the name of the file or the current date or time from the computer system, and they display this information in a document. FrameMaker provides a variety of predefined system variables. The end user or template designer cannot delete these variables or add new ones in a document, but can edit the definitions of the variables.

You can specify which variable to use in an instance of a system variable element. When an end user inserts the element in a document, the element automatically uses the specified system variable and displays the information from the variable. The format rule for a system variable is binding. The user cannot change a system variable element to another variable.

To specify a system variable, insert a `SystemVariableFormatRule` element in the system variable element's definition (after any attribute definitions). Then insert and define context elements as necessary. For each context, insert a `UseSystemVariable` child element and then a nested child element to specify a variable, or insert a `DefaultSystemVariable` child element to use the `FilenameLong` variable. For example:

Element (System Variable): Date
System variable format rule
1. **If context is:** TitlePageDate
Use system variable: Current Date (Long)
Else
Use system variable: Current Date (Short)

These are the system variables available:

Variable	Default definition	Example
CreationDateLong	<$monthname> <$dayname>, <$year>	May 15, 1995
CreationDateShort	<$monthnum>/<$daynum>/ <$shortyear>	5/15/95
CurrentDateLong	<$monthname> <$dayname>, <$year>	July 6, 1995
CurrentDateShort	<$monthnum>/<$daynum>/ <$shortyear>	7/6/95
CurrentPageNum	<$curpagenum>	3
FilenameLong	<$fullfilename>	/usr/devguide/ objrules
FilenameShort	<$filename>	objrules
ModificationDateLong	<$monthname> <$dayname>, <$year>, <$hour>:<$minute00> <$ampm>	August 1, 1995, 2:30 pm
ModificationDateShort	<$monthnum>/<$daynum>/ <$shortyear>	8/1/95
PageCount	<$lastpagenum>	18
RunningHF1	<$paratext[Title]>	The Turbulent Oceans
RunningHF2	<$paratext[Heading1]>	Threat of Extinction
RunningHF3	<$marker1>	Inspection Checklist
RunningHF4	<$marker 2>	Drawing Objects
TableContinuation	(Continued)	(Continued)
TableSheet	(<Sheet <$tblsheetnum> of <$tblsheetcount>)	(Sheet 1 of 2)

Some of the system variables, such as those for running headers and page numbers, are used most often on master pages. If your end users apply structure only to body pages, you will not use these variables in an EDD.

If you do not specify a variable for a system variable element, the element uses the variable `FilenameShort`.

FrameMaker does not have a variable element for user variables. An end user can insert a user variable as an object in a container element.

Debugging object format rules

After writing object format rules, you should try them out by importing the EDD into a sample document and inserting new instances of the elements. If any objects are not formatted the way you expect, check the EDD for these errors:

- Typing errors in the names of table formats or cross-reference formats in formatting specifications, or in the element tags or sibling indicators in context specifications
- Incorrect child elements in the formatting specifications for graphics formats, marker types, equation sizes, or system variables
- Duplicated context specifications
- Format rule clauses that are not in specific-to-general order

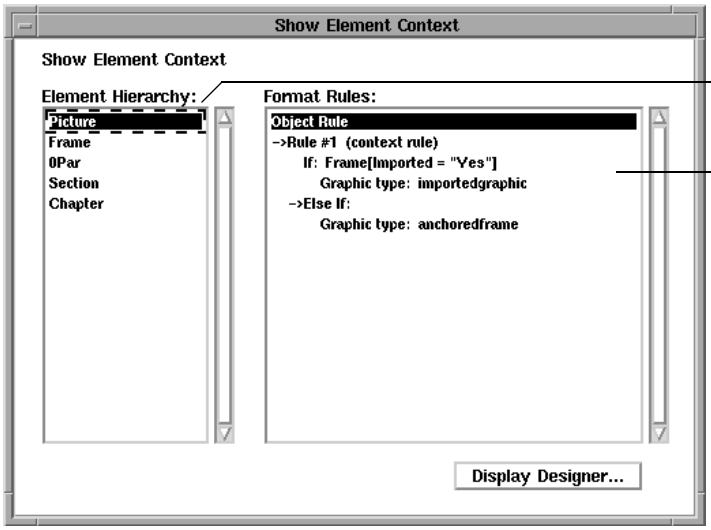
You might also need to look at formats in the document or the template to see if they are defined the way you expect. Check the definitions in these places:

- Table formats in the Table Designer (Table menu)
- Cross-references formats in the Cross-Reference dialog box (Special menu)
- Equation sizes in the Equation Sizes dialog box (Equations pop-up menu in the Equations palette)
- System variables in the Variable dialog box (Special menu)

If FrameMaker identifies any problems when you import an EDD, it produces a log file of warnings and errors. For information on how to work with this file, see "Log files for imported element definitions" on page 104.

As you examine the formatted contents of the document, you can use the Show Element Context dialog box to find out what object format rule is being applied to a particular instance of an element. To open the dialog box, choose Show Element Context from the File>Developer Tools submenu.

Show Element Context lists the hierarchy of the current element on the left, beginning with the current element on top. If you select an element in the list, the right side of the dialog box shows that element's format rules. The rule clause that applies to the particular element instance has an arrow pointing to it.



If the selected element is a container, table title, table cell, or footnote, the scroll box on the right shows text format rules instead of object format rules. For an example of this dialog box with text rules, see "Debugging text format rules" on page 166.

To display the formatting properties for a table format, select the format in the list on the right and click Display Designer.

Part III Translating between markup data and FrameMaker

Part III provides details of the model FrameMaker uses for translating between markup data and FrameMaker documents and on how you can modify this translation. You should be familiar with the material in Part I, "Developing a FrameMaker structure application" before using the material in this part.

This part of the manual includes:

- Chapter 13, "Introduction to Translating between Markup Data and FrameMaker"

  Describes the types of modifications you can accomplish with read/write rules. The chapter also contains a simple example of using rules to create a structure application.

- Chapter 14, "Read/Write Rules and Their Syntax"

  Describes the mechanics of writing read/write rules and the document in which you define them.

- Chapter 16, "Translating Elements and Their Attributes"
  Chapter 17, "Translating Entities and Processing Instructions"
  Chapter 18, "Translating Tables"
  Chapter 19, "Translating Graphics and Equations"
  Chapter 20, "Translating Cross-References"
  Chapter 21, "Translating Variables and System Variable Elements"
  Chapter 22, "Translating Markers"
  Chapter 23, "Translating Conditional Text."
  Chapter 24, "Processing Multiple Files as Books"

  Contain information on how FrameMaker translates various specific markup and FrameMaker constructs. Each of these chapters describes the software's default translation followed by modifications you can make to that translation with read/write rules.

- Chapter 25, “Read/Write Rules Summary”

  Contains a summary of the read/write rules provided for translating between various markup and FrameMaker constructs.

- Chapter 26, “Read/Write Rules Reference”

  Contains an alphabetical list of reference material on all read/write rules.

13 Introduction to Translating between Markup Data and FrameMaker

FrameMaker can read and write markup data without any help from you. However, its default translation of markup constructs may not be suitable for your DTD. In such cases, you have to write a structure application to modify the translation.

In this chapter

This chapter talks about the information your application typically contains. In the outline below, click a topic to go to its page.

Descriptions of the types of modifications to the default translation that you can make:

- "What you can do with read/write rules," next
- "What you can do with structure API clients" on page 202

An example to illustrate these points:

- "A detailed example" on page 203

Discussion of locking XML and FrameMaker files:

- "Opening XML documents" on page 207

What you can do with read/write rules

Read/write rules are needed for a variety of reasons. Most rules perform tasks falling in one of these categories:

Working with special constructs Because markup doesn't standardize a model for constructs such as tables, graphics, or cross-references, their handling is unique to each DTD. When creating a DTD from an EDD, FrameMaker makes assumptions about how to translate these FrameMaker constructs to markup elements and attributes. Your application may need only to modify the default translation in minor ways using rules. On the other hand, when creating an EDD from a DTD, FrameMaker cannot recognize these constructs, so your rules need to be more extensive.

Renaming elements and attributes FrameMaker element tags and attribute names are frequently more descriptive than their XML or SGML counterparts and you might use a read/write rule to establish a correspondence between names. For example, you could use a rule to change the FrameMaker tag `Employee Name` to the XML generic identifier `ename`.

Treating special characters In SGML and XML you can use an entity to represent special characters. You might use rules to translate such entities.

> **XML:** The XML specification establishes *predefined* character entities for reserved characters such as angle brackets. By default FrameMaker translates these characters correctly on import and export. However, you will need to specify read/write rules to export such characters as a numeric references (`&#XXXX;`).

Making information implicit Not all information relevant to one representation is relevant to the other. For example, you may have a type of table that always has two columns. In XML you have an element for this table and do not need to explicitly indicate in the document instance that the table has two columns. Nevertheless, in FrameMaker you need to specify a number of columns when you insert the table. By default, the software writes an attribute containing the number of columns on export; you can choose instead to specify this information in a read/write rule.

Unwrapping structure You may have levels of hierarchical element structure in one representation that are unnecessary in the other. If you are translating documents from one system to the other and don't need to translate those documents back again, you may decide to simplify the element hierarchy.

Other rules There are a few rules that aren't used for any of the above purposes. These rules are discussed in detail in Chapter 26, "Read/Write Rules Reference." Sections of the chapter that describe examples of such rules include:

- "character map" on page 369 describes the rule that tells FrameMaker how to map between characters in the SGML and FrameMaker character sets.
- "external dtd" on page 392 describes the rule that determines whether to include an external DTD subset or to copy the specified DTD into the internal DTD subset. By default, FrameMaker includes an external DTD subset.
- "line break" on page 443 describes the rule that tells FrameMaker how to interpret line breaks when importing an XML or SGML document and when to generate line breaks when exporting a FrameMaker document.
- "write structured document" on page 466 and "write structured document instance only" on page 466 describe the rules that tell FrameMaker when saving as markup whether it should write only a markup document instance or an entire document.

What you can do with structure API clients

In situations in which read/write rules are insufficient to express the proper translation between markup and FrameMaker, you can create a structure API client. You use the Frame Developer's Kit (FDK) to modify the import or export of markup documents. You cannot use the FDK to directly modify how the parser creates a DTD or an EDD.

Working with the FDK requires C and FDK programming skills. The FDK allows you an arbitrary amount of control over FrameMaker's processing. For this reason, there is no way to briefly describe the possibilities here.

During import, your FDK functions can inspect or modify the structure, format, or content of the FrameMaker document being constructed. FDK functions can process attribute values, processing instructions, and some entity references encountered in the imported markup document.

During export, your functions can inspect the structure, format, and content of the exported FrameMaker document. They can inspect and modify every generated portion of the markup document before it is actually written, and they can create new text (data and markup) to be inserted into the XML or SGML document.

This manual does not describe how to create a structure API client, although it does occasionally mention situations in which a client is appropriate. For information on creating structure API clients, see the *Structure Import/Export API Programmer's Guide*.

A detailed example

To give you a more concrete feeling for translating between markup and FrameMaker, this section contains an example of a DTD fragment and the corresponding EDD fragment, rules for translating between them, and snippets of related SGML and FrameMaker documents.

This section does not explain how to create the statements shown here. Later chapters provide the details.

DTD fragment

Here is a fragment of the DTD:

```
<!--Shorthand for name of the system being described-->
<!ENTITY plan CDATA "Transportation 2000">

<!--Entities for special characters.-->
<!ENTITY oquote SDATA "open-quote">
<!ENTITY cquote SDATA "close-quote">

<!--Parameter entity for text that can appear in a paragraph.-->
<!ENTITY % text "(#PCDATA | xref)*">

<!--Basic element structure-->
<!ELEMENT section - - (head, (para | list)*, section*)>
<!ELEMENT (head, para) - O (%text;)>
<!ELEMENT list - - (item+)>
<!ATTLIST list type (bullet | number) bullet>
<!ELEMENT item - O (%text;, list?)>
```

```
<!--Allow * as the start-tag of an item element-->
<!ENTITY strtitem STARTTAG "item">
<!SHORTREF listmap "*" strtitem>
<!USEMAP listmap list>

<!--Cross-references-->
<!ELEMENT xref - O EMPTY>
<!ATTLIST xref id IDREF #IMPLIED>
```

This DTD fragment defines several entities. The `plan` entity is shorthand for the string "Transportation 2000"; this makes it easy for an end user to change the plan name in one place. The `oquote` and `cquote` entities are system-specific characters—the directional open and close quotation marks. Finally, `%text` is a parameter entity that simplifies entering a frequently used portion of the content model in the DTD.

Next are several fairly straightforward element definitions for the `section`, `head`, `para`, `list`, and `item` elements. The `list` element has an attribute presumably directing a structure application on the appropriate formatting of an instance of the element. There are also several declarations that allow users to use an asterisk (*) as markup to indicate the start of an `item` element in a `list` element. The `xref` element and its attributes describe cross-references.

Document instance

A portion of a document instance written using this DTD looks like this:

```
<section>Summary of &plan; Plan Elements
<head>Highway System
<para>
A base network of roads for people and goods movement designed to
operate at maximum efficiency during off-peak and near capacity in
peak hours. Elements include freeways, expressways, and major
arterials.
<list type=number>
*Completion of Measure &oquote;A&cquote; program for Routes 101, 85,
237
*Emphasis on Commuter Lanes and bottleneck improvements including
new and upgraded interchanges
*Capacity improvements in 101 and Fremont/South Bay Corridors
*Operational improvements including signal synchronization and
freeway surveillance
</list>
</list>
```

EDD fragment

Here is an EDD fragment you might produce from the DTD fragment:

Element (Container): Section
General rule: Head, (Para | List)*, Section*
Text format rules
Element paragraph format: body

Element (Container): Head
General rule: (<TEXT> | Xref)*
Text format rules
1. Count ancestors named: Section
If level is: 1
Use paragraph format: head1
Else:
Use paragraph format: head2

Element (Container): Para
General rule: (<TEXT> | Xref)*

Element (Container): List
General rule: Item+
Attribute list
1. Name: Type **Choice** **Optional**
Choices: Bullet, Number
Default: Bullet

Element (Container): Item
General rule: (<TEXT> | Xref)*, List?
Text format rules
1. If context is: {first} < List [Type = “Number”]
Use paragraph format: 1Step
Else, if context is: List [Type = “Number”]
Use paragraph format: Step
Else, if context is: List < Item
Use paragraph format: 2Bullet
Else:
Use paragraph format: Bullet

Element (Cross-reference): Xref
Initial cross-reference format
1. In all contexts.
Use cross-reference format: NumOnly

Each element definition specifies formatting appropriate for that element and its descendants. The `Section` element specifies body as the paragraph format for all text unless a descendant specifies otherwise. The `Head` element specifies Head1 as the paragraph format if it occurs as the child of a single `Section` element and Head2 if it

occurs as the child of more than one Section element. The Item element bases it's formatting on the value of the Type attribute of the parent List element.

Formatting and read/write rules

When creating an EDD from a DTD, FrameMaker automatically generates most of these FrameMaker element definitions (other than their formatting rules) from the corresponding markup declarations without any intervention. You only need to specify format rules, and one read/write rule that states that the markup element xref should become the special cross-reference element type.

There is more interesting information in the DTD than just the element structure, however. By default, FrameMaker creates a variable with the variable text "Transportation 2000" corresponding to the plan entity. This translation is probably what you want. It also creates variables for the oquote and cquote entities, with the variable text "open-quote" and "close-quote", respectively. This is probably not what you want for those entities. You can supply rules to have the oquote and cquote entities become the directional open and close quotation characters.

The plan variable and the information about the oquote and cquote entities aren't part of the EDD. The variable definition is stored directly in a FrameMaker document that uses that entity; the information about the oquote and cquote entities remains solely in the rules, although a FrameMaker document created from a markup document that uses those entities contains open and close quotation characters, as appropriate.

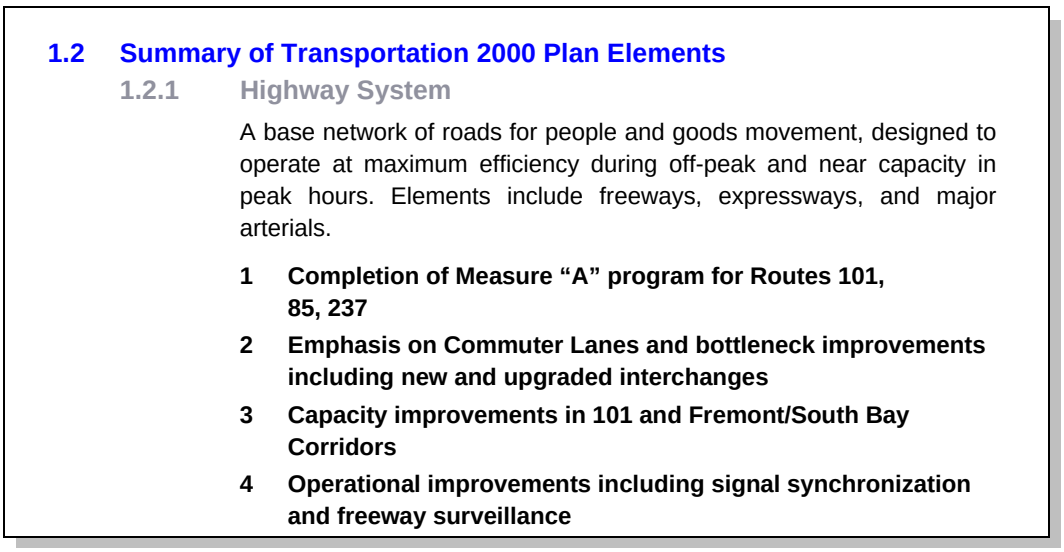
So, the complete set of read/write rules you'll need is as follows:

```
/* Change the xref element to a special element type. */
element "xref" is fm cross-reference element;

/* Translate SDATA entities directly into the FM character set. */
entity "oquote" is fm char "\“";
entity "cquote" is fm char "\”";
```

FrameMaker document

With these rules and an appropriate template, FrameMaker translates the earlier SGML markup to the following in a FrameMaker document window:



Given the above structured FrameMaker document, on export FrameMaker produces SGML markup equivalent to that shown earlier. Without a structure API client, it doesn’t produce short references or markup minimization.

Opening XML documents

When you open an SGML document, FrameMaker creates a structured FrameMaker document. It proposes a name for this new file by replacing any extension in the name of the SGML document with `.fm` (and appending `.fm` to the end of the SGML document’s filename if that name has no extension). It uses the proposed name to label the document window, but allows you to change the name before saving the new document. When you save the document or use the File > Save As command, by default, FrameMaker saves the document as a FrameMaker document.

By default, when you open an XML document, however, FrameMaker retains the original name. When you save the imported document, FrameMaker saves the file back to XML, under the original filename. This default behavior resembles corresponding behavior for working with FrameMaker documents. It differs in the following ways:

- If you have not yet saved the file, you can use the File > Revert to Saved command to discard changes that have been made to a FrameMaker document. This command cannot be used with XML documents.
- You can use File > Preferences > General to set an automatic save option. When this option is set, FrameMaker automatically saves edited documents at specified time intervals. FrameMaker does automatically save XML documents, but it saves them as FrameMaker documents. If you need to open the autosave file for an XML document, perhaps after a system crash or power failure, you can save it as XML.

- If you have enabled Network File Locking through File > Preferences > General, FrameMaker locks both FrameMaker and XML documents that you are editing to prevent other users from modifying files you have open. However, it uses different mechanisms for locking FrameMaker and XML documents. On UNIX platforms, other applications may ignore UNIX conventions and still change a file that is open in FrameMaker.

You can save an opened XML document as a FrameMaker document. You can work with the resulting FrameMaker document as you would any other FrameMaker document. When you have finished editing, you can save the result as XML.

Two configuration settings allow you to modify the default behavior. Since these settings are made in `maker.ini` on Windows and in the `Maker` resource file on UNIX, they apply to all your structure applications. To change them, you must exit from FrameMaker, change the setting, and then restart FrameMaker. Both settings are made in the `Preferences` section of the platform-specific configuration file:

- `TreatXMLAsXML` can be set to either `On` or `Off`; the default is `On` (on UNIX the setting is called `treatXmlAsXmlDocuments` and the possible values are `True` and `False` with a default of `True`.). When it is `On`, FrameMaker retains the name of an imported XML document as described above. When it is `Off`, FrameMaker uses the extension `.fm` for imported XML documents and by default saves them as FrameMaker documents. In this case, the file naming and file typing for imported XML parallels that for SGML.
- `LockXmlFiles` can also be set to either `On` or `Off` with a default of `On`. (On UNIX, the setting is called `Maker.lockXmlFiles` and the possible values are `True` and `False` with a default of `True`.) If the setting is `On` (`True` on UNIX), FrameMaker locks XML documents that are opened when `TreatXMLAsXml` is `On`. If `LockXmlFiles` is `Off`, FrameMaker does not lock open `XML` documents.

14 Read/Write Rules and Their Syntax

Read/write rules are your primary device for modifying FrameMaker's default translations.

In this chapter

This chapter describes the syntax of the rules and how you create them. Chapter 26, "Read/Write Rules Reference," describes each of the rules in detail. In the outline below, click a topic to go to its page.

General description of a read/write rules document:

- "The rules document" on page 209

Discussion of the significance of the order of rules in a rules document:

- "Rule order" on page 210

Discussion of the syntax of individual rules:

- "Rule syntax" on page 211
- "Case conventions" on page 212
- "Strings and constants" on page 212
- "Comments" on page 214

Splitting the rules for an application between several files:

- "Include files" on page 214

Names you shouldn't use for your FrameMaker elements:

- "Reserved element names" on page 214

User interface commands you use when developing a read/write rules document:

- "Commands for working with a rules document" on page 215

The rules document

You create a FrameMaker document or an ASCII file containing read/write rules. The software uses this rules document to modify its import and export processing of markup data.

Assuming that you create a FrameMaker document for your rules, you enter a set of rules into the main flow on the body pages of the document. The document can be structured or unstructured and can use any element definitions or formatting properties desired.

FrameMaker reads the content of the document and not its structure. Keywords in rules cannot include non-breaking hyphens.

> **Important:** Eight-bit characters, greater than 0x80, are not supported in filenames occurring within read/write rule files. For portability, avoid using those characters in filenames within read/write rule files.

To create a new read/write rules document, choose File>Developer Tools>New Read/Write Rules. This command creates a new rules document, using the template it finds in `$STRUCTDIR/default.rw`. For information on the location of `$STRUCTDIR`, see "Location of structure files" on page 49.

The first rule in a rules document must be:

```
fmsgml version is "7.0";
```

The string only includes one level of "dot" releases. For example, you would use the string `"5.5"` even though the product version may be an incremental release above 5.5, such as 5.5.1 or 5.5.6.

Most rules are relevant all the time—for example, a rule to convert the general entity `pname` to the FrameMaker variable `Product Name`. Some, however, are only relevant to certain situations. For example, a rule to export graphics into external entities named `graphic1.mif`, `graphic2.mif`, and so forth is relevant only when exporting FrameMaker documents. It is not relevant when exporting an EDD or when importing markup documents or DTDs, because these situations do not require generating entity names for graphics.

There is so much overlap in the rules that FrameMaker adopts the strategy of having one document specify all rules, rather than using a separate document for each type of operation: import of a DTD, import of an XML or SGML document, export of an EDD, or export of a FrameMaker document. Most rules are expressed from the markup perspective; the *element* of an `element` rule is the XML or SGML generic identifier, not the FrameMaker element tag. The only exceptions to this convention are rules for FrameMaker constructs that have no counterpart in markup. For example, a rule might specify dropping FrameMaker markers on export.

A single rules document can also contain rules that do not apply to a particular document. This allows the same rules document to be used with several related document types, even if rules exist for constructs used in only some of them. Markup and FrameMaker provide the same flexibility, by allowing a DTD or EDD to define constructs that are never used. Because unused rules can also result from spelling errors in generic identifiers and in the names of other constructs, FrameMaker issues warning messages in the log file when rules refer to constructs that do not exist.

Rule order

Although the first rule in a read/write rules file must specify the product version, in general the order in which other rules appear is not significant. The only time rule order is significant

is when multiple rules apply equally to the same situation. In such cases, the software uses the rule that appears *first* in the document. For example, assume you have these rules:

```
element "list" is fm element "List";
element "list" is fm element "Procedure";
element "proc" is fm element "Procedure";
```

These rules say that two different FrameMaker elements correspond to the same markup element, and two markup elements correspond to the same FrameMaker element. FOr example, if you import an XML document to FrameMaker with these rules, all occurrences of the XML `list` element become `List` elements, and occurrences of the `proc` element become `Procedure` elements. If you export a FrameMaker document to XML, both `List` and `Procedure` elements become `list` elements. And no `proc` elements are created. The result is that if you start with an XML document containing a `proc` element, import that document to FrameMaker, and then export the result back to XML, the original `proc` element becomes a `list` element.

On the other hand, the order of the following rules does not matter:

```
fm attribute "XRefLabel" drop;

element "chapter"
   attribute "xreflab" is fm attribute "XRefLabel";
```

The first of these rules tells the software to discard the FrameMaker `XRefLabel` attribute in any element in which it occurs. The second rule tells the software to translate the FrameMaker `XRefLabel` attribute to the XML `xreflab` attribute within the context of the `Chapter` element. The effect of these two rules is to discard the `XRefLabel` attribute when it occurs in any element other than the `Chapter` element.

Rule syntax

Rules use a syntax similar to C language syntax, in which each rule begins with a keyword and ends either with a brace-enclosed group of subrules or with a semicolon. Names of markup and FrameMaker constructs, such as elements, attributes, and entities, are represented by strings. For information on strings, see “Strings and constants” on page 212. Rules can be *nested;* that is, they can occur inside one another. A rule nested inside another is called a *subrule*. One that is not nested is called a *highest-level rule*. A typical rule containing a brace-enclosed set of two subrules is:

```
element "vex" {
   is fm element "Verbatim Example";
   attribute "au" is fm attribute "author";
}
```

In this example, `element` is the highest-level rule. The `is fm element` and `attribute` rules are subrules of the `element` rule. The `is fm attribute` rule is a subrule of both the `element` and `attribute` rules.

A single subrule can appear at the end of a rule without the enclosing braces. For example:

```
element "list" {
   attribute "indent" drop;
}
```

is equivalent to

```
element "list" attribute "indent" drop;
```

Null rules, consisting simply of a semicolon, can appear wherever a rule is allowed.

Case conventions

For keywords of the read/write rules syntax, case is not significant. For example, the following are equivalent for rules in a structure application:

```
fm version is "7.1";
FM Version is "7.1";
```

To improve readability, this manual sometimes uses mixed case for keywords.

The case of FrameMaker element tags and other FrameMaker names is always significant. Thus, the following rules are not equivalent:

```
fm element "Default Element" drop;
fm element "default element" drop;
```

The significance of the case of SGML names is dependent on the NAMECASE parameter of the SGML declaration. For XML the case is always significant, and your rules must match the case used in the XML markup. For more information, see "Naming elements and attributes" on page 231.

Strings and constants

You use strings to specify FrameMaker element tags, generic identifiers in markup, attribute names, attribute values, entity names, notation names, and so on.

String syntax

Strings are enclosed in matching straight or directional double quotation marks. Strings cannot extend across more than one paragraph.

To incorporate special characters into strings, use the following conventions:

- `\x` followed by one or two hexadecimal digits is interpreted as a hexadecimal character code.
- `\0` (backslash zero) followed by one or two octal digits is interpreted as an octal character code.

- A backslash followed by one, two, or three other digits is interpreted as a decimal character code.
- A backslash immediately preceding any character other than a digit or the letter *x* indicates that the following character is part of the string. In particular, a double quotation mark or backslash can be preceded by a backslash to enter it into a string.

Constant syntax

Some rules and parameter literals for entities accept a single constant character code instead of a one-character string with a character code. In these cases, the constant character code takes a slightly different format than it does within a string. In particular:

- A number starting with `0` (zero) followed by other digits is interpreted as an octal number.
- A number starting with `0x` (zero x) followed by other digits is interpreted as a hexadecimal number.
- Any other number is interpreted as a decimal number.

Variables in strings

FrameMaker defines a small set of variables that can be used within strings in particular rules. These variables have the following syntax:

`$(`*`variable_name`*`)`

where *`variable_name`* must be a legal variable name. An example is the variable `$(Entity)`.

The following table illustrates the conventions for entering strings. The strings occur within a rule in which the variable `$(Entity)` is interpreted as `CHIPS`:

Syntax within rule	Interpreted string
`"Chips"`	Chips
`"Potato Chips"`	Potato Chips
`"\""`	"
`"\\"`	\
`"a$(entity)z"`	aCHIPSz
`"a\$(entity)z"`	a$(entity)z
`"a$entityz"`	a$entityz
`"\xa7Chips"`	ßChips

For more information on entities, see Chapter 17, “Translating Entities and Processing Instructions.”

Comments

Comments can appear anywhere in a rules document where white space is permitted, except within quoted strings. Comments, like those in C code, are surrounded by the delimiters `/*` and `*/`. Tables, graphics, and equations can appear within comments but are erroneous elsewhere in a rules document.

Include files

You can use the C notation for include files in a rules document. For example, assume you have the following line in a paragraph by itself in a rules document:

```
#include "fname"
```

FrameMaker processes the file named *fname* as though its contents were inserted in place of the include directive. The syntax of the filename is device-dependent.

You can specify a search path for include files in the `structapps.fm` file. The default search path for an include file consists of the directory containing the original rules document and the directory `$STRUCTDIR/isoents/`, where `$STRUCTDIR` is as defined in "Location of structure files" on page 49. In addition, these directories are added to the end of the search path you specify in `structapps.fm`.

If you plan to use the same rules on different systems, avoid specifying directory names in the `#include` directive. Instead, let FrameMaker find files in different directories through a search path.

Important: File paths must be specified in the syntax that is native to the platform on which you are running FrameMaker. Also, for Windows platforms, pathname tokens are separated by a backslash character ("\"). When specifying a Windows path, you must escape the backslash character. For example, the following specify the same location on Unix, Macintosh, and Windows platforms:

- Unix
  `$STRUCTDIR/isoents`
- Macintosh
  `$STRUCTDIR:isoents`
- Windows
  `$STRUCTDIR\\isoents`

For information on working with the `structapps.fm` file, see "Application definition file" on page 50.

Reserved element names

You may use any legal name for an XML or SGML element. If there are no rules to the contrary, FrameMaker assumes most markup elements correspond to FrameMaker elements. The following generic identifiers are used by the default declarations for

translating FrameMaker cross-references, equations, footnotes, graphics, or tables to markup: CROSSREF, EQUATION, FOOTNOTE, GRAPHIC, TABLE, TITLE, HEADING, BODY, FOOTING, ROW, and CELL.

SGML: FrameMaker uses the following entity names for translating FrameMaker system variables to SGML: fm.pgcnt, fm.ldate, fm.sdate, fm.lcdat, fm.scdat, fm.lmdat, fm.smdat, fm.lfnam, fm.sfnam, fm.tcont, and fm.tsht.

If your DTD uses any of these names for another purpose, you will need to write a rule to provide alternate names for the constructs provided by the default declarations.

XML: In the absence of read/write rules, FrameMaker exports system variables as text. To preserve data as system variables for import and export, declare an empty element in your DTD for each system variable your document uses, and specify rules to convert the element to a system variable.

Commands for working with a rules document

The File>Structure Tools menu provides two commands for working with a read/write rules document: New Read/Write Rules and Check Read/Write Rules.

- New Read/Write Rules

  To create a new rules document, choose New Read/Write Rules. This command uses the rules template in $STRUCTDIR/default.rw, if there is one. For information on the $STRUCTDIR directory, see "Location of structure files" on page 49.

- Check Read/Write Rules

  To verify the correctness of a rules document, choose Check Read/Write Rules. This command works with the current document. That document must be a read/write rules file that has been saved to a file. When you choose the Check Read/Write Rules command, you get a dialog box asking you to pick the application to check the rules against. You can choose an application for this command, or check the rules without the added context of an application. If the application you choose specifies a different read/write rules document, that rules document is ignored for the purposes of this command.

  The Check Read/Write Rules command checks for syntax errors in rules and also reports semantic errors on the basis of the DTD and the FrameMaker template specified in the selected application, if there is one. Errors are reported in a log file. Note that your read/write rules document can refer to markup constructs not mentioned in the current DTD—for details, see "The rules document" on page 209.

For information on adding a DTD file or rules document to an application, see "Defining an application" on page 53.

15 Saving EDD Formatting Information as a CSS Stylesheet

Typically, an EDD contains a significant amount of formatting information. FrameMaker uses this information to map formatting properties to elements. This mapping depends on the given element's tag, its position in the document structure, and its values for specified attributes.

In XML this type of formatting information is typically expressed in a cascading stylesheet (CSS). However, if an XML document specifies a CSS FrameMaker does not use the CSS when it opens the XML. Instead, FrameMaker uses the formatting that is specified in the template—definitions in the format catalogs, plus the formatting rules specified in the template's EDD.

When you save a FrameMaker document as XML, you may want to generate a CSS for the XML or you may want to use a CSS that was provided with the original XML document. To make that decision, you should understand how to specify the actions FrameMaker will take, and you should understand how FrameMaker generates a CSS.

In this chapter

This chapter discusses how to generate a CSS in FrameMaker, and how EDD formatting information translates to a CSS. In the outline below, click a topic to go to its page.

Descriptions of the types of modifications to the default translation that you can make:

- "Default translation," next
- "Generating a CSS" on page 224

Default translation

Even though an EDD and a CSS both assign formatting properties to elements according to similar criteria, there are differences between the two. For example, in an EDD the indents of an element's text are specified absolutely; in CSS child elements inherit the indents of their parent elements, and additional indents accumulate with the inherited values.

In addition, the CSS statements generated by FrameMaker may not work on all browsers. For example, `AcrossAllColumns` and `AcrossAllSideheads` in an EDD translate to `display:compact` in CSS—this CSS statement is currently supported by the Opera6 browser, but not by NetScape Navigator 6.x or Internet Explorer 6.x.

Comparison of EDD format rules and CSS

Following is a table of format rule properties that can be used in an EDD, and how FrameMaker translates them to CSS. The table also indicates any CSS statements that are not supported by a specific browser. The browsers to be noted in this list are NetScape Navigator version 6.0 or later (NS), Internet Explorer version 6.0 or later (IE), and Opera version 6.0 or later (OP).

Font properties

Property in EDD	CSS statement	Values (and comments)	Browser support
Font size	font-size		ALL
Angle	font-style		ALL
Variation	font-variant		ALL
Weight	font-weight		ALL
Stretch	font-stretch		ALL
Case	text-transformation	uppercase or lowercase	ALL
Color	Color		ALL
Changebars	UNSUPPORTED		
CombinedFont	font-family		ALL
Family	font-family		ALL
OffsetHorizontal	UNSUPPORTED		
OffsetVertical	UNSUPPORTED		
Outline	UNSUPPORTED	No equivalent in CSS	
Overline	text-decoration	overline	ALL
PairKerning	UNSUPPORTED	No equivalent in CSS	
Shadow	text-shadow	Uses offsets of 0, 0 and same color	ALL
SizeChange	font-size	No equivalent in CSS	ALL
Spread	UNSUPPORTED	No equivalent in CSS	
SpreadChange	UNSUPPORTED	No equivalent in CSS	
StretchChange	UNSUPPORTED	No equivalent in CSS	
StrikeThrough	text-decoration	line-through	ALL
Superscript	vertical-align	Super	ALL
Subscript	vertical-align	Sub	ALL
Tracking	UNSUPPORTED		
TrackingChange	UNSUPPORTED		
Underline	text-decoration	underline	ALL
Language	UNSUPPORTED	No equivalent in CSS	

Property in EDD	CSS statement	Values (and comments)	Browser support
Tsume	UNSUPPORTED		

Indent properties

Property in EDD	CSS statement	Values (and comments)	Browser support
FirstIndent FirstIndentRelative FirstIndentChange	text-indent	Padding and border margins set to zero—NS and IE may not support negative indents	ALL
LeftIndent LeftIndentChange	margin-left	Padding and border margins set to zero	ALL
RightIndent RightIndentChange	margin-right	Padding and border margins set to zero	ALL
LineSpacing Height HeightChange	line-height		ALL
SpaceAbove SpaceAboveChange	margin-top		ALL
SpaceBelow SpaceBelowChange	margin-below		ALL
AsianSpacingroperties WesternAsianSpacing AsianAsianSpacing	UNSUPPORTED		ALL
Punctuation	UNSUPPORTED		ALL

Pagination properties

Property in EDD	CSS statement	Values (and comments)	Browser support
KeepWithNext	page-break-after:avoid		ALL
KeepWithPrevious	page-break-before:avoid		ALL
AcrossAllCols	display:compact		OP
AcrossCosSideHeads	display:compact		OP
InColumn	display:block		
RunInHead	display:run-in		NONE
SideHead	display:compact or float:left/right		ALL

Property in EDD	CSS statement	Values (and comments)	Browser support
DefaultPunctuation	after content		OP
Alignment	UNSUPPORTED	No equivalent in CSS	
Anywhere	UNSUPPORTED	Support unnecessary - this is unmodified pagination	
TopOfColumn	page-break-before:auto		ALL
TopOfLeftPage	page-break-before:right		ALL
TopOfPage	page-break-before:always		ALL
TopOfRightPage	page-break-before:left		ALL
WidowOrphanLines	widws/orphans		ALL
PgfAlignment	text-align		ALL

Autonumbering properties

Property in EDD	CSS statement	Values (and comments)	Browser support
AutoNumberFormat	list-style-type or counter	The choice of list-style-type or counter depends on the complexity of the autonumbering	ALL
AutoNumCharFormat	UNSUPPORTED	No equivalent in CSS—Generates char styles for element:before	OP
Position	beginning	Always position autonums at the beginning	ALL

Advanced properties

Property in EDD	CSS statement	Values (and comments)	Browser support
FrameAbove	border-top	FrameAbove and FrameBelow translate as border lines, only—the actual graphic in the reference frame does not translate to CSS	ALL
FrameBelow	border-bottom		ALL
LetterSpacing	letter-spacing		ALL
WordSpacing	word-spacing		ALL
Hyphenation	UNSUPPORTED	No equivalent in CSS	

Table cell properties

Property in EDD	CSS statement	Values (and comments)	Browser support
CellMargins	padding	In the EDD you can specify values for Change, Custom, or FromTableFormatPlus—padding only supports Custom	OP, NS
VerticalAlignment	vertical-alignment		ALL

Miscellaneous rules, properties, and elements

Property in EDD	CSS statement	Values (and comments)	Browser support
FirstParagraphRules	UNSUPPORTED		
LastParagraphRules	UNSUPPORTED		
Prefix/Suffix rules	before/after rules		OP
FormatChangeList	UNSUPPORTED	No equivalent in CSS	
Limits	UNSUPPORTED		
Rubi elements	UNSUPPORTED		
Use Paragraph Format		Generates appropriate CSS statements for the paragraph format	

EDD selectors This table lists element selectors for the element named XYZ

Selector in EDD	CSS selector	Browser support (and comments)
All contexts	XYZ{...}	ALL
List < Heading	Heading>List>XYZ	NS, OP
(List | Heading)	List > XYZ{...} Heading > XYZ{...}	NS, OP
{first} < Numbered	Numbered: first-child	NS
XYZ {after ABC}	ABC + XYZ	NS, OP
{middle} < Numbered	ABC > XYZ[fmcssattr="1"]	NS, OP
{last}< ABC	ABC > XYZ[fmcssattr="1"]	NS, OP
{notfirst} < ABC	ABC > XYZ[fmcssattr="1"]	NS, OP
{notlast} < ABC	ABC > XYZ[fmcssattr="1"]	NS, OP
only	ABC > XYZ[fmcssattr="1"]	NS, OP
{before ABC}	XYZ + ABC	NS, OP
{after ABC}	ABC + XYZ	NS, OP

Selector in EDD	CSS selector	Browser support (and comments)
{between ABC QRS}	ABC + XYZ + QRS {}	NS, OP
{any}	*	ALL
XYZ[att != val]	ABC > XYZ[fmcssattr="1"]	NS, OP
XYZ[att >= val]	ABC > XYZ[fmcssattr="2"]	NS, OP
XYZ[att <= val]	ABC > XYZ[fmcssattr="3"]	NS, OP
XYZ[att <val]	ABC > XYZ[fmcssattr="4"]	NS, OP
XYZ[att > val]	ABC > XYZ[fmcssattr="5"]	NS, OP
count ancestors named XYZ level is: 3	XYZ XYZ XYZ	NS, OP Supports up to 10 ancestors
count ancestors named XYZ stop counting at ancestor ABC If level is 2	ABC XYZ XYZ	NS, OP

Differences in translation

The format rules in an EDD express some things in ways that don't translate directly to CSS. The following section describes these differences, and how FrameMaker resolves them.

Right and left indents

In FrameMaker the right and left indents are expressed as absolute distances from the right or left edge of the text frame. For example, a section head may have a left indent of 24 pt from the text frame, and the body paragraphs following the head may have left indents of 48 pt. In FrameMaker these indents are expressed absolutely, 24 pt and 48 pt, respectively.

In CSS an element inherits indents from its ancestors. Using the example above, you would express the following:

```
Head{margin-left:24 pt;}
```

```
Head>Body{margin-left:24 pt;}
```

When FrameMaker generates a CSS, it translates its absolute indents to values for inherited margins that give you the same result.

Context-specific format rules

An EDD can include context-specific format rules that can define one or more possible contexts, with formatting changes for each context. The contexts are expressed in separate if, else/if, and else clauses. These clauses can include sibling indicators such as {middle}, {last}, {notfirst}, etc. For more information, see "Context-specific rules" on page 134.

CSS has no equivalent for these sibling indicators. For example, assume the following format rule for the Para element:

Element (Container): Para
General rule: <TEXT>
Text format rules
1. **If context is:** Section{notfirst}
Default font properties
Angle: Italic

In this case, the Para element uses italic text unless its parent Section element is the first of a group of siblings. In that case, the Para element will be italic.

There is no mechanism in CSS to express this selection. To reproduce the formatting, FrameMaker writes a special attribute to the affected Para element (in XML), and refers to that attribute in the CSS. These attributes have the reserved name of *fmcssattr*.

For example, to reproduce the above effect in CSS and XML, FrameMaker writes the following for the affected element:

para fmcssattr=”1”>

and it writes the following CSS statements:

para[fmcssattr=1] {font-style:italic;}

You should consider the following things when FrameMaker performs these actions:

- This is a modification to the XML, and it adds information that is specific to the FrameMaker product. This is deprecated in markup, because an important aspect of markup is that it doesn’t depend on any single product.
- If the XML complies with a DTD, the added attributes are likely to make the XML invalid.
- Unless you included these attributes in your EDD, you will get an invalid FrameMaker document when you import the XML that includes these attributes. To avoid this result, you can either define the attributes in the affected elements, or you can define read/write rules to drop these attributes on read.

Because of these issues, you can disable this effect in your application definition that you specify in `structapps.fm`. To disable this effect, you set *AddFmCSSAttrToXml* to *Disable* in your application definition.

If Else rules

The use of an Else statement in the EDD translates into two CSS statements. For example, with this statements in an EDD:

```
If Xref < Para Then color = Blue Else color = Black
```

the resulting CSS looks like:

```
Para > Xref{color: blue}
Xref {color = black;}
```

This will usually give appropriate results. There are circumstances when the EDD uses the order of processing to limit execution, which result in CSS that can't make the same distinction. For example, if the EDD includes:

```
If Emphasis < Emphasis color = blue Else if Emphasis Italic = Yes
```

the resulting CSS looks like this:

```
Emphasis > Emphasis {color: Blue;}
Emphasis {font-style: italic;}
```

In the EDD, if the first clause is true then the second clause will not take effect—Emphasis will be blue but not italic. With the CSS processing, the same element will appear both blue and italic.

Unsupported rules and statements

The following rules and statements in an EDD cannot translate to CSS. When it encounters these constructs, FrameMaker continues to generate the CSS, but ignores the constructs:

- Subrules and nested rules
- AND statements in selectors
- Table cell indents
- First and last paragraph rules

Generating a CSS

When working with FrameMaker you can choose to generate a CSS on command or automatically whenever you save a FrameMaker document as XML.

Generating a CSS on command

The simplest method to generate a CSS is to choose *File Structure* > *Tools Generate* > *CSS2*. With this command, FrameMaker generates a CSS from the EDD information that is stored in the current document's element catalog. FrameMaker saves the CSS with the file name and location that you specify.

> **Important:** You should not perform this action with a template file. To correctly generate a CSS, FrameMaker must know which element is used as the root element for the document. An EDD can define a number of `ValidHighestLevel` elements, but the CSS must refer to only one. FrameMaker relies on actual use in the document to determine which is the root element. Since a template usually has no content, FrameMaker cannot make the determination. You should generate a CSS from a document that has content—either use an existing document, or insert a `ValidHighestLevel` element in your template before generating the CSS from it.

After you generate a CSS in this way, the stylesheet is not necessarily used by any XML documents. If you specify the same path and file name as an existing CSS, and that CSS is used by an XML document, the following will result:

- The new CSS will overwrite the older one.
- Any XML documents that specify a CSS of that path and file name will now use the new CSS.

Generating a CSS on Save As XML

When editing XML documents, you must control whether FrameMaker generates a CSS every time you save a document as XML. For example, if an XML document specifies an existing CSS, you may want to retain that specification each time you save the document as XML. On the other hand, if you generate a CSS on Save As XML, FrameMaker will change the CSS specification to use this generated CSS.

Specifying how FrameMaker generates a CSS

You specify how FrameMaker will generate a CSS by inserting elements in the structure application definition. These elements are all children of the *Stylesheets* element. You specify these elements as follows:

- To direct FrameMaker to generate a CSS, set *GenerateCSS2* to *Enable*.
- To specify a path for the stylesheet, provide a URI in *StylesheetURI*. If you don't provide a URI but have enabled *GenerateCSS2*, FrameMaker will save the generated stylesheet at the same location as the resulting XML document.
- To have FrameMaker support EDD context selectors in the CSS via a reserved attribute, set *AddFmCSSAttrToXml* to *Enable*.
- To direct FrameMaker to save the generated CSS in a specific location, provide a URI in the *StylesheetURI* element. Otherwise, FrameMaker saves a generated CSS in the same location as the ML document.
- To direct FrameMaker to save XML with a new stylesheet declaration that uses the URI specified in your structure application, set *RetainStylesheetPIs* to *Disable*.
- To ensure that the stylesheet declaration in the exported XML is the same as it was when you imported the XML, set *RetainStylesheetPIs* to *Enable*.
- To specify the type of stylesheet you are using, provide a string in the *StylesheetType* element. Currently, FrameMaker can only generate a CSS stylesheet. Even if you are not generating a stylesheet, you should provide a stylesheet type in case other processes rely on that data. For example, a structure API client may vary its processing depending on the value you provide here.

Stylesheet specifications in a structure application

You specify stylesheet parameters in the application definitions file, `structapps.fm`. For an XML application, you can include the following elements.

Stylesheets The parent element for the CSS specification. It contains *CssPreferences*, *RetainStyleSheetPIs*, and *XmlStylesheet* elements.

CssPreferences Specifies whether or not to generate a CSS, and whether to generate special attributes in the XML. It contains *GenerateCSS2* and *AddFmCSSAttrToXML* elements.

GenerateCSS2 Specifies whether FrameMaker will generate a CSS when you save the document as XML. It can be set to *Enable* or *Disable*. When this is set to Enable, FrameMaker generates a CSS. If a path is provided in StylesheetURI, FrameMaker saves the stylesheet to that location, with that filename. Otherwise, it saves the stylesheet to the same location as the XML document with a filename *xmldoc*.css, where *xmldoc* is the name of the XML document you're saving.

AddFmCSSAttrToXml Specifies whether FrameMaker will write instances of the *fmcssattr* attribute to elements in the XML document. It can be set to *Enable* or *Disable*. An EDD can include context selectors as criteria to assign format rules. CSS has no equivalent to this. When this is set to *Enable*, FrameMaker uses the *fmcssattribute* in certain elements so the CSS can achieve the same formatting as the EDD.

RetainStylesheetPIs Specifies whether FrameMaker will retain the stylesheet declaration for import and export of XML. It can be set to *Enable* or *Disable*. When this is set to *Enable*, FrameMaker does the following:

- On import, it stores the XML document's stylesheet PI as a marker in the FrameMaker document.
- On export, it writes the content of stylesheet PI marker in the resulting XML document.

XmlStylesheet Specifies the type of stylesheet and the URI for the stylesheet. It contains the *StylesheetType* and *StylesheetURI* elements.

StylesheetType Specifies the type of stylesheet. It contains a string for the stylesheet type. Currently, you can specify CSS (upper or lower case) or XSL (upper or lower case). If you specify XSL, FrameMaker will not generate a stylesheet.

StylesheetURI Specifies the URI for the stylesheet. It contains a string; for example, /$STRUCTDIR/xml/xhtml/app/xhtml.css.

16 Translating Elements and Their Attributes

Elements and their attributes are the fundamental components of both FrameMaker and markup documents.

Most information in this chapter is “bidirectional.” It is about translations that can be recognized and performed when both reading and writing markup documents and DTDs, and the examples apply to both cases. You can start with a DTD containing element and attribute declarations and produce the corresponding FrameMaker element definition, or you can start with an EDD containing the FrameMaker element definition and produce the corresponding XML or SGML element and attribute declarations.

In this chapter

This chapter gives you an overview of how FrameMaker translates between its representation of elements and attributes and corresponding representations in markup. In the outline below, click a topic to go to its page.

How FrameMaker translates elements by default:

- “Translating model groups and general rules” on page 228
- “Translating attributes” on page 229
- “Naming elements and attributes” on page 231
- “Inclusions and exclusions” on page 233
- “Line breaks and record ends” on page 233

Some ways you can change the default translation:

- “Renaming elements” on page 234
- “Renaming attributes” on page 235
- “Renaming attribute values” on page 235
- “Translating a markup element to a footnote element” on page 236
- “Translating a markup element to a Rubi group element” on page 237
- “Changing the declared content of a markup element associated with a text-only element” on page 238
- “Retaining content but not structure of an element” on page 238
- “Retaining structure but not content of an element” on page 239

- "Formatting an element as a boxed set of paragraphs" on page 240
- "Suppressing the display of an element's content" on page 240
- "Discarding a markup or FrameMaker element" on page 240
- "Discarding a markup or FrameMaker attribute" on page 241
- "Specifying a default value for an attribute" on page 242
- "Changing an attribute's type or declared value" on page 243
- "Creating read-only attributes" on page 244
- "Using markup attributes to specify FrameMaker formatting information" on page 244

This chapter gives you information about elements and their attributes as a general class. For information on how to translate elements and attributes used for a particular purpose, see Chapters 18 through 24.

Default translation

The basic representations of elements and attributes in markup and in FrameMaker are very similar, facilitating the translation in both directions. This section provides information on how the parts of element and attribute definitions translate by default.

> **Important:** For importing and exporting XML, FrameMaker has limited support of double-byte characters in markup tokens such as GIs and attribute names. For more information, see "Supported characters in element and attribute names" on page 26and "Specifying the character encoding for XML files" on page 68.
>
> SGML does not support double-byte characters in markup tokens.

Translating model groups and general rules

The general rule of a FrameMaker element uses a syntax based on SGML model groups. FrameMaker uses the delimiter strings defined in the SGML reference concrete syntax for connectors and occurrence indicators. For example, suppose you have the following declaration:

```
<!ELEMENT lablist - - (head, par?, item+)>
```

By default, the corresponding FrameMaker element definition is:

Element (Container): Lablist
General rule: Head, Par?, Item+

When FrameMaker converts a DTD to an EDD, it performs the following translations:

- The token #PCDATA in a content model translates to the token <TEXT> in a FrameMaker general rule.
- A content model of ANY translates to the FrameMaker general rule <ANY>.

- A declared content of either CDATA or (for SGML) RCDATA translates to the FrameMaker general rule <TEXTONLY>.
- A declared content of EMPTY translates to the FrameMaker general rule <EMPTY>.

When FrameMaker converts an EDD to a DTD, it performs the reverse translations. In the case of a <TEXTONLY> general rule it produces a declared content of PCDATA for XML, and a declared content of RCDATA for SGML.

Translating attributes

In markup, attribute declarations for an element occur in a separate attribute definition list declaration. In FrameMaker, the attribute definitions for an element are directly part of that element's definition.

For example, assume you have the following declarations in SGML:

```
<!ELEMENT lablist - - (head, par?, item+)>

<!ATTLIST lablist
   id ID #IMPLIED
   type (num | bul) bul
   sec (u | s | t) #REQUIRED>
```

By default, these translate to the following element definition in FrameMaker:

Element (Container): Lablist
General rule: Head, Par?, Item+
Attribute list

1. Name: Id	**Unique ID**	**Optional**
2. Name: Type	**Choice**	**Optional**
Choices: Num | Bul		
Default: Bul		
3. Name: Sec	**Choice**	**Required**
Choices: U | S | T		

Note that the first two markup attributes become optional attributes in FrameMaker. In addition, the interpreted attribute value specification given in the default value for the markup type attribute translates to a default value in FrameMaker.

In general, any markup attribute that is not a required attribute (that is, doesn't use the #REQUIRED declared value) becomes an optional attribute in FrameMaker. If the markup attribute has an attribute value specification, the interpreted version of that value becomes the attribute's default value in FrameMaker.

By default, a small number of markup attributes translate to formatting properties in FrameMaker. This happens only in the case of graphics, equations, and tables that use a standard representation recognized by FrameMaker. For information on these attributes, see Chapter 18, "Translating Tables," and Chapter 19, "Translating Graphics and Equations."

When FrameMaker writes a FrameMaker document as XML or SGML, it writes markup attribute specifications for attributes with an explicitly supplied value. Such explicit values may be entered directly by the end user, created by the FrameMaker cross-reference facility, or supplied by an FDK client.

Conversely, on import of a markup document, if an imported element doesn't include a value for an attribute, then FrameMaker only supplies a value if the EDD includes a default value for the attribute—otherwise FrameMaker doesn't supply a value for such an attribute.

Attribute types and declared values

FrameMaker has a set of attribute types that correspond to the declared values for attributes in XML or SGML, but neither is a subset of the other. That is, multiple declared values in markup can become the same FrameMaker attribute type, and conversely multiple FrameMaker attribute types can become the same markup declared value. (XML and SGML do not define the term *attribute type*. Loosely speaking, you can think of an attribute's declared value as its type.)

When you create an EDD from a DTD or a DTD from an EDD, FrameMaker uses a default translation between attribute types and declared values.

> **XML:** XML does not support all the declared values that are supported in SGML. The following tables identify the declared values that are valid only in SGML. For XML, substitute these declared values as follows:
>
> - `NAME`, `NUMBER`, and `NUTOKEN` to `NMTOKEN`
> - `NAMES`, `NUMBERS`, and `NUTOKENS` to `NMTOKENS`

On import, FrameMaker makes the following conversions in the absence of read/write rules:

Declared Value in Markup		FrameMaker Attribute Type
`CDATA`		`String`
`ENTITY`		`String`
`ENTITIES`		`Strings`
`ID`		`UniqueID`
`IDREF`		`IDReference`
`IDREFS`		`IDReferences`
`NAME`	SGML-only	`String`
`NAMES`	SGML-only	`Strings`
`NMTOKEN`		`String`
`NMTOKENS`		`Strings`
`NUMBER`	SGML-only	`Integer`
`NUMBERS`	SGML-only	`Integers`
`NUTOKEN`	SGML-only	`String`

Declared Value in Markup		FrameMaker Attribute Type
NUTOKENS	SGML-only	Strings
NOTATION		Choice
Name token group		Choice

On export, FrameMaker makes the following conversions in the absence of read/write rules:

FrameMaker Attribute Type	Declared Value in Markup
String	CDATA
Strings	CDATA
Integer	**SGML:** NUMBER if values are restricted to be positive, CDATA otherwise. **XML:** NMTOKEN
Integers	**SGML:** NUMBERS if values are restricted to be positive, CDATA otherwise **XML:** NMTOKENS
Real	**SGML:** NUMBER if value is restricted to be positive, CDATA otherwise **XML:** NMTOKEN
Reals	**SGML:** NUMBERS if values are restricted to be positive, CDATA otherwise **XML:** NMTOKENS
UniqueID	ID
IDReference	IDREF
IDReferences	IDREFS
Choice	Name token group

Naming elements and attributes

Legal names in FrameMaker and in markup are different. FrameMaker element tags and attribute names can be longer than allowed by the SGML reference concrete syntax and can contain characters (such as the $ character) that are not allowed by the naming rules of the reference concrete syntax.

In XML a name can contain Unicode characters. For information about FrameMaker support for unicode in names, see "Supported characters in element and attribute names" on page 26.

If you create an EDD by importing an SGML DTD, you won't have trouble with illegal names. Any name that is legal in the SGML reference concrete syntax is legal in FrameMaker. However, if you create an SGML DTD by exporting an EDD, you are likely to

encounter naming difficulties unless you have adhered to SGML naming rules in your EDD. You will need to write rules to reflect the differences.

Likewise, if you create an EDD by importing an XML DTD, you will have no problems. If you create an XML DTD by exporting an EDD, you should not encounter problems with name length, but you will need to ensure the element and attribute names use only legal characters. Otherwise you will need to write rules to reflect the differences.

Important: For importing and exporting XML, FrameMaker has limited support of double-byte characters in markup tokens such as GIs and attribute names. For more information, see "Supported characters in element and attribute names" on page 26and "Specifying the character encoding for XML files" on page 68.

SGML does not support double-byte characters in markup tokens.

If you do not specify the name for an element or attribute when translating between markup and FrameMaker, FrameMaker must pick a name to use. The chosen name has the same characters as the original. For XML, FrameMaker preserves the case of names on import and export.

For SGML the case conventions are often different than case conventions in FrameMaker. To ensure that the case of a name is appropriately chosen, FrameMaker uses this information:

- If specified, an `element` rule that explicitly translates a generic identifier to a FrameMaker element tag

  If an `element` rule gives both markup and FrameMaker names for an element, the software uses the names and their case as specified.

- If specified, an `attribute` rule that explicitly translates a markup attribute name to a FrameMaker attribute name

  If an `attribute` rule gives both markup and FrameMaker names for an attribute, the software uses the names and their case as specified.

- The NAMECASE parameter in the SGML declaration

  If the NAMECASE parameter states that names are case sensitive, then the software preserves the case of all characters on import and export. If the NAMECASE parameter states that names are not case sensitive (as it does for general names in the reference concrete syntax, which FrameMaker uses by default), then:

  - On import, the software converts a generic identifier to an element tag that has an initial capital letter followed by lowercase characters. For example, if a generic identifier is written in the SGML document as `part`, `Part`, PART, or `pArt`, it becomes the FrameMaker element tag `Part`. The software performs the same conversion for attribute names.
  - On export, the software converts a FrameMaker element tag to a generic identifier that has all lowercase characters. So, each of the FrameMaker element tags `part`, `Part`,

PART, and pArt, become the generic identifier part. The software performs the same conversion for attribute names.

Important: If the NAMECASE parameter states that names are not case sensitive (NAMECASE GENERAL YES), the same characters with different capitalization in FrameMaker export as the same element GI or attribute name. For example, FrameMaker elements named Part and part both export as the SGML element part, resulting in an error. If the SGML declaration states that names *are* case sensitive, these FrameMaker elements become distinct SGML elements as intended.

Inclusions and exclusions

XML: The XML specification does not allow inclusions and exclusions. The following discussion pertains to SGML, only.

Just as SGML content models can specify exceptions to the model group, FrameMaker element definitions can specify exceptions to the general rule. The following is an element declaration that lists inclusions:

```
<!ELEMENT book - - (front, body, back) +(index)>
```

On import, FrameMaker produces this FrameMaker element definition:

Element (Container): Book
General rule: Front, Body, Back
Inclusions: Index

Similarly, the following is an element definition that lists exclusions:

Element (Container): Appendix
General rule: Title, Section+
Exclusions: List, Table

On export, FrameMaker produces this SGML element declaration:

```
<!ELEMENT appendix - - (title, section+) -(list, table)>
```

Line breaks and record ends

You can control the behavior of markup record ends and FrameMaker line breaks on import and export of documents.

By default, when importing a markup document, FrameMaker treats a data record end within a text segment as a space. It ignores record ends in other locations. You can change this behavior for record ends that are not ignored by the XML or SGML parser.

By default, when exporting a FrameMaker document, FrameMaker behaves as follows:

- When exporting the text of a paragraph, it ignores line breaks. It includes a space separating the two words on either side of a line break.

- It generates a record end at the end of every paragraph and flow in the FrameMaker document.

For information on how you can change this behavior, see “line break” on page 443.

Modifications to the default translation

You can use rules to change the translation of elements and their attributes. The most common change is to rename them upon transfer between FrameMaker and markup.

The following sections describe some of the possible modifications to the default translations. Your situation may require different rules or a structure API client, or you may use these rules in ways not discussed in these sections; the cross-references point to related information. In addition, the other translation chapters point to further information on particular element types.

For a summary of read/write rules relevant to translating all elements and attributes, see “All Elements” on page 355. For information on writing structure API clients, see the*Structure Import/Export API Programmer’s Guide*.

Renaming elements

There are many reasons you may choose to rename elements when translating between markup and FrameMaker. As already mentioned, the FrameMaker naming conventions are less restrictive than those of SGML. If you wish to take advantage of this, you could rename your elements on import and export.

> **Important:** Note that read/write rules do not support double-byte characters, so you cannot use rules to convert element names that have double-byte characters.

The general form of the rule for renaming an element is:

```
element "gi"
   is fm element "fmtag";
```

where *gi* is a generic identifier and *fmtag* is a FrameMaker element tag. For example, if you have an XML element `par` the FrameMaker element’s name is by default `par`. To change this default behavior, you could use this rule:

```
element "par"
   is fm element "Paragraph";
```

With this rule, import of a document or DTD translates the XML element `par` as the FrameMaker element `Paragraph`. Conversely, export of a FrameMaker document or EDD translates the FrameMaker element `Paragraph` as the XML element `par`.

For information on these rules, see “element” on page 376 and “is fm element” on page 419.

Renaming attributes

Just as you can choose to rename elements, you can choose to rename their attributes when translating between markup and FrameMaker. You can rename an attribute either for all elements in which it occurs or only for a particular attribute. To do so, you use one of these rules:

```
attribute "attr"
   is fm attribute "fmattr";

element "gi"
   attribute "attr"
       is fm attribute "fmattr";
```

> **Important:** Note that read/write rules do not support double-byte characters, so you cannot use rules to convert attribute names that have double-byte characters.

where *attr* is a markup attribute name, *fmattr* is a FrameMaker attribute name, and *gi* is a generic identifier.

The first form renames the attribute no matter in which element it occurs. For example, if you have an XML attribute `sec`, the FrameMaker attribute's name is by default `sec`. If this attribute occurs in several elements and you want the same alternate name for all those elements, you could use this rule:

```
attribute "sec" is fm attribute "Security";
```

With this rule, import of a document or DTD translates the markup attribute `sec` as the FrameMaker attribute `Security`. Conversely, export of a FrameMaker document or EDD translates the FrameMaker attribute `Security` as the markup attribute `sec`.

Sometimes you may wish to rename an attribute differently for different elements. For example, assume you use an attribute named `ref` for the elements `article` and `xref`. If these attributes have different meanings you may want to choose different names for them in FrameMaker. You could use these rules:

```
element "article"
   attribute "ref" is fm attribute "Reference";

element "xref" {
   is fm cross-reference element "XRef";
   attribute "IDref" is fm attribute "ID";
}
```

For information on these rules, see "attribute" on page 367, "element" on page 376, "is fm attribute" on page 414, and "is fm cross-reference element" on page 418.

Renaming attribute values

As with attributes and elements, there are also reasons you may choose to rename the members of a name token group when translating between markup and FrameMaker. You

can rename their values either for all occurrences, or within a single context. To do so, you use one of these rules:

```
value "val" is fm value "fmval";

attribute "attr" value "val" is fm value "fmval";

element "gi" attribute "attr" value "val" is fm value "fmval";
```

where *val* is a name token, *fmval* is a string, *attr* is an XML attribute, and *gi* is a generic identifier.

For example, assume the attribute `color` is used for many elements with the values `r`, `b`, and `g`. You could write this rule to rename the values in the same way for all occurrences of the `color` attribute:

```
attribute "color" {
   value "r" is fm value "Red";
   value "b" is fm value "Blue";
   value "g" is fm value "Green";
}
```

For information on these rules, see "element" on page 376, "attribute" on page 367, "value" on page 464, and "is fm value" on page 439.

Translating a markup element to a footnote element

Many documents require footnotes to provide additional information. Handling of footnotes is primarily a formatting issue. Because of this, markup does not directly address them. FrameMaker, however, has a special type of element for creating and formatting footnotes.

Because FrameMaker maintains footnotes as elements, there is no problem when writing a FrameMaker document as markup data. A footnote element becomes an XML or SGML element. However, if you have a markup document or DTD with an element that you want to treat as a footnote in FrameMaker, you must write a rule to indicate this.

For example, to translate the markup element `fn` as a footnote element named `Footnote` in FrameMaker, use this rule:

```
element "fn" is fm footnote element "Footnote";
```

In FrameMaker, a footnote element is always a footnote element. That is, you can't import the same element as a footnote in some contexts and as a normal paragraph in other contexts. If you have a single markup element that can be formatted as both a footnote and as a normal paragraph, you'll need to translate the same markup element to two different FrameMaker elements. This translation cannot be accomplished with read/write rules. In this case, you'll need to write a structure API client.

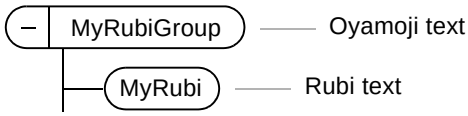
For information on these rules, see "element" on page 376 and "is fm footnote element" on page 421. For information on writing structure API clients, see the *Structure Import/Export API Programmer's Guide*.

Translating a markup element to a Rubi group element

Documents that include Japanese text most likely require Rubi to express the pronunciation of certain words. Handling of Rubi is primarily a formatting issue. Because of this markup does not directly address Rubi. However, FrameMaker has a special type of element for creating a *Rubi group*. The Rubi group includes a range of text for the Oyamoji, and it assigns Rubi text to the Oyamoji.

Since the Rubi group is an element, when writing a FrameMaker document as markup a Rubi Group element and its contents become markup elements. However, if you have markup document or DTD with an element you want to translate as a Rubi group in FrameMaker, you must use read/write rules to indicate this.

The minimal structure for a Rubi group is:



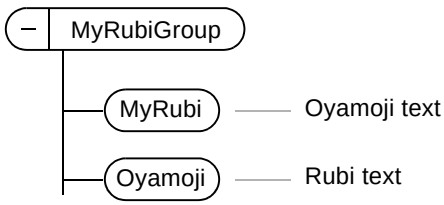
The XML specification suggests a representation of a Rubi group that includes Oyamoji in a container element as follows:

```
<ruby>
   <rb>Oyamoji text</rb>
   <rt>Rubi text</rt>
</ruby>
```

To translate this Rubi group, use these rules:

```
element "ruby" is fm rubi group element "MyRubiGroup";
element "rb" is fm rubi element "MyRubi";
element "rt" is fm rubi element "Oyamoji";
```

If the element named MyRubiGroup allows the Oyamoji element in its content rule (and if the EDD includes a definition for the Oyamoji element) the markup will translate to FrameMaker data with the following structure:



Another typical representation of a Rubi group in markup is:

```
<rubigroup rubi = "Rubi text">Oyamoji text</rubigroup>
```

FrameMaker does not support direct translation of this representation. To create a Rubi group when the Rubi text is represented as an attribute, you'll need to write a structure API client.

If you have an XML or SGML element that is used as a Rubi group in some cases and some other text range in other cases, you'll need to translate the same markup element to two different FrameMaker elements. This translation cannot be accomplished with read/write rules. In this case, you will need to write a structure API client.

For information on these rules, see "element" on page 376, "is fm rubi group element" on page 432, and "is fm rubi element" on page 432. For information on writing structure API clients, see the *Structure Import/Export API Programmer's Guide*.

Changing the declared content of a markup element associated with a text-only element

XML: RCDATA is not valid in XML, so you should never find yourself importing XML that has an element declared as RCDATA. When saving to XML, if an element is declared as <TEXTONLY>, FrameMaker translates its general rule as (#PCDATA).

SGML: By default, FrameMaker translates an element with a general rule of <TEXTONLY> as an SGML element with declared content RCDATA.

An RCDATA element can include entity and character references. If you want your markup element not to allow entity references, you must manually edit the DTD and change the declared content of the element to CDATA.

Note, however, that the FrameMaker editing environment cannot enforce this change for you. Your end user can insert user variables or special characters into a <TEXTONLY> element, even if the corresponding markup element is CDATA. The entity or character references normally used to represent such data are not recognized within CDATA content.

If the end user uses such constructs and exports that document to markup, the software reports an error in the log file. If your application contains CDATA elements, you should make your end users aware of this restriction.

Retaining content but not structure of an element

There may be elements in your DTD that you do not need in your EDD. Similarly, there can be elements in your EDD that you do not need in your DTD. In this case, you can write rules to retain the contents of an element while removing the element itself. If you do this, however, any attributes associated with the element will be discarded. Also, if your document structure must be preserved on a round trip to and from markup, you must write a structure API client to recreate the element on transfer in the other direction.

To remove element structure, but not content, use one of these rules:

```
element "gi" unwrap;

fm element "fmtag" unwrap;
```

For example, assume you have these element declarations in your DTD:

```
<!ELEMENT memo - - (top, body)>
<!ELEMENT top O O (to, from, date)>
<!ELEMENT body O O (par+)>
```

You can choose to omit the `top` and `body` elements on import to FrameMaker. Use these rules:

```
element "top" unwrap;
element "body" unwrap;
```

Import of the DTD produces this element definition:

Element (Container): Memo
General rule: ((To, From, Date), Par+)

For information on these rules, see "element" on page 376, "fm element" on page 397, and "unwrap" on page 461.

If you use one of these rules, you may want to use the `preserve fm element definition` rule to aid maintenance of your EDD. If you do not, updating your EDD on the basis of a modified DTD may change your EDD definitions in ways you do not intend. For information on this rule, see "preserve fm element definition" on page 448.

For information on structure API clients, see the *Structure Import/Export API Programmer's Guide*.

Retaining structure but not content of an element

You can use some elements in markup or FrameMaker as place holders for information generated by a tool. For example, FrameMaker can generate a table of contents for a document. So when you import a markup element containing a table of contents you may choose to have the software generate a new version of the content instead of including the text of the element. To discard an element's content but retain its structure, use one of these rules:

```
element "gi"
   reader drop content;

element "gi"
   writer drop content;
```

For example, assume your DTD has this element declaration:

```
<!ELEMENT toc - - (#PCDATA)>
```

In markup, you depend on an application to create the appropriate content for the `toc` element. In FrameMaker, you want to use its tools for generating a table of contents. To do so, first write the rule:

```
element "toc" reader drop content;
```

To generate the appropriate content for the element on import to FrameMaker, you have two choices. You can require the end user to manually call the Generate command, or you can create an FDK client to generate the table of contents as the last step in importing the document. With this rule, when you export the FrameMaker document to markup the `toc` element retains its content in the exported document.

For information on these rules, see "element" on page 376, "drop content" on page 375, "reader" on page 453, and "writer" on page 467. For information on creating an API client, see the *FDK Programmer's Reference*.

Formatting an element as a boxed set of paragraphs

The formatting associated with your documents may require that the paragraphs in an element appear in a completely or partially boxed area. In FrameMaker, you must have a table element to get this formatting. For information on how to translate a single markup element as a one-cell table, see "Using a table to format an element as a boxed set of paragraphs" on page 293.

Suppressing the display of an element's content

You may have a markup element you want to retain but whose content you do not want displayed when you translate markup documents containing the element to documents in FrameMaker. The simplest method of accomplishing this is to translate the element to a marker element and its content to marker text.

For example, if you have a `comments` element that should not print in the document but whose content you do not want to lose, you can use these rules:

```
element "comments" {
   is fm marker element;
   marker text is content;
}
```

For more information on marker elements, see Chapter 22, "Translating Markers." For more information on these rules, see "element" on page 376, "is fm marker element" on page 423, and "marker text is" on page 444.

Alternatively, you may not want to create a marker element from your markup element. If you do not, there are other ways you could suppress display of the element's content, but these methods require you to write a structure API client. For example, you could have an API client attach a condition tag to the element and then hide that condition tag. For information on creating structure API clients, see the *Structure Import/Export API Programmer's Guide*.

Discarding a markup or FrameMaker element

You can use markup elements for purposes that have no counterpart in FrameMaker, and the reverse is also true. For example, assume you use an element `break` to indicate in

your markup that a page or line break is desired. You can discard this element when the markup document is imported to FrameMaker and have a structure API client insert the necessary break; there is no need to mark the break with an element. To drop an element on import, use this rule:

```
element "break" drop;
```

As another example, you may have a table that always has the same column headings. For this reason, you choose not to include an element for the column headings in the XML or SGML representation. If your table is to have column heading in FrameMaker, however, the headings must be in elements; you would discard these elements on export using this rule:

```
fm element "ColumnHead" drop;
```

The `drop` rule actually discards the data—if you use the `drop` rule and want the element to be restored when you are translating a document in the other direction, you must write a structure API client.

For information on these rules, see "element" on page 376, "fm element" on page 397, and "drop" on page 373.

If you use either of these rules, you may want to use the `preserve fm element definition` rule to aid in the maintenance of your EDD. If you do not, updating your EDD on the basis of a modified DTD may change your EDD definitions in ways you do not intend. For information on this rule, see "preserve fm element definition" on page 448.

For information on creating a structure API client, see the *Structure Import/Export API Programmer's Guide*.

Discarding a markup or FrameMaker attribute

If you use attributes in markup for purposes that have no counterpart in FrameMaker, or for purposes for which you must supply a structure API client, you can choose to discard these attributes. To discard an attribute for all elements that use it, use one of these rules:

```
attribute "attr" drop;
fm attribute "attr" drop;
```

where *attr* is an XML, SGML, or FrameMaker attribute name.

To discard the attribute for a single element, use one of these rules:

```
element "gi" attribute "attr" drop;
element "gi" fm attribute "attr" drop;
```

where *gi* is a generic identifier and *attr* is an XML, SGML, or FrameMaker attribute name.

For example, assume that you have these declarations in SGML:

```
<!ELEMENT par - O (#PCDATA)>
<!ATTLIST par fontsize NUMBER 10>
```

The fontsize attribute indicates formatting for the element. You'll have to specify that information in some other way, perhaps with a format rule in your EDD or with a structure API client. To have the software discard the attribute fontsize on import, use this rule:

```
element "par" attribute "fontsize" drop;
```

If you use the drop rule and want the attribute to be restored when you are translating a document in the other direction, you must write a structure API client.

For information on these rules, see "attribute" on page 367, "drop" on page 373, "element" on page 376, and "fm attribute" on page 396. For information on writing format rules, see Chapter 10, "Text Format Rules for Containers, Tables, and Footnotes." For information on creating a structure API client, see the *Structure Import/Export API Programmer's Guide*.

Specifying a default value for an attribute

An attribute in markup can be implicit, in which case end users don't need to provide a value for it. For example, your DTD may declare the following:

```
<!ATTLIST (appendix | chapter | reference | section)
    label (ch | app) #IMPLIED>
```

In markup, the appendix, chapter, reference, and section elements can include either ch or app as values for the label attribute, but they don't have to. You may want to set up your EDD to format these elements based on the value of the label attribute. Further, you may want to use a default attribute value to format these elements in case the markup doesn't include a value for the label attribute.

FrameMaker provides a rule to set up a default value for attributes in the EDD when you import a DTD. To do this at the highest level, for all elements that use the relevant attribute, use this rule:

```
attribute "attr" implied value is "val";
```

where *attr* is a markup attribute and *val* is the proposed value for that attribute.

To specify a value only for the attribute within a particular markup element, use this rule:

```
element "gi" attribute "attr" implied value is "val";
```

where *gi* is a generic identifier, *attr* is an attribute in the element, and *val* is the proposed value for that attribute.

In the above example, assume the default value for label should be ch for most of these elements, but app for an appendix element. You can specify values to use in all cases with these rules:

```
attribute "label" implied value is "ch";
element "appendix" attribute "label" implied value is "app";
```

Note that this rule is for importing DTDs and exporting EDDs. In FrameMaker, a default attribute value can only be specified in the EDD, so this rule has no effect when importing

an XML or SGML instance, or when exporting a FrameMaker document. Also, the default value is used only to initiate formatting. FrameMaker does not actually provide an attribute value for the element. When you export the FrameMaker document, the software does not generate an attribute value where there previously was none.

For more information, see "Default value" on page 176 and "Default translation" on page 228 For information on these rules, see "element" on page 376, "attribute" on page 367, and "implied value is" on page 406.

Changing an attribute's type or declared value

Declared values for attributes in markup, and FrameMaker attribute types are different. In the absence of read/write rules, FrameMaker must make decisions on how to translate between them. The default translations are listed in "Attribute types and declared values" on page 230. You can change the attribute type created for an EDD or the declared value for a DTD by using the following rule:

```
[structdv | xmldv] attribute "attr"
       is fm [read-only] [fmtype] attribute "fmattr"
       [range from a to b];
```

where *structdv* is an SGML declared value, *xmldv* is an XML declared value, *attr* is a markup attribute name, *fmtype* is a FrameMaker attribute type, *fmattr* is a FrameMaker attribute name, and *a* and *b* are numbers.

SGML: The possible values for *structdv* are:

cdata	id	name	number
entity	idref	names	numbers
entities	idrefs	nmtoken	nutoken
group	notation	nmtokens	nutokens

XML: The possible values for *xmldv* are:

cdata	id	notation
entity	idref	idrefs
entities	idrefs	notation
group		

The possible values for *fmtype* are:

string	integers	unique-id
strings	real-number	id-reference
choice	real-numbers	id-references
integer		

For example, assume your DTD defines `size` as an attribute of type `CDATA`. If you know that values of this attribute always represent real numbers, you can use this rule:

```
cdata attribute "size" is fm real-number attribute;
```

As another example, if your SGML DTD defines `perc` as an attribute of type NUMBER, you can use this rule to change the attribute name:

```
NUMBER attribute "perc" is fm integer attribute "Percentage";
```

Be careful with the use of this rule. FrameMaker allows you to translate any declared value to any attribute type. For instance, in the `size` example, if you are wrong that the attribute is always a real number, your end users may encounter errors reading their SGML documents.

For information on read-only attributes, see "Creating read-only attributes," next. For information on these rules, see "attribute" on page 367 and "is fm attribute" on page 414.

Creating read-only attributes

FrameMaker allows you to define read-only attributes. A read-only attribute is one whose value cannot be changed by an end user.

For example, assume the following situation. You use a structure API client to extract elements from an XML database that you then combine into a FrameMaker document. In the database, the elements are identified by keys. You need to be able to store an edited version of these elements back in the database, so you need to retain the keys to use. To do so, you have your structure API client store them as the value of an attribute of the element when read into FrameMaker. Also, you do not want your users to change the value of the attribute, as that would invalidate the key. To prevent that, you make the attribute a read-only attribute.

Another common use of read-only attributes is for `ID` and `IDREF` attributes associated with cross-references. Since FrameMaker can automatically maintain these attribute values for your end users, you may choose not to let them manually modify the values.

To specify an attribute to be read-only, use the following rule:

```
attribute "attr" is fm read-only attribute ["fmattr"];
```

where *`attr`* is an attribute name in markup, and *`fmattr`* is a FrameMaker attribute name.

For information on these rules, see "attribute" on page 367 and "is fm attribute" on page 414.

Using markup attributes to specify FrameMaker formatting information

A common use of attributes is for storing formatting information. FrameMaker does not recognize markup attributes used in this way by default. However, your EDD can contain format rules that do use attributes in this way.

To create the appropriate translation, you may need to use read/write rules to rename attribute values. In addition, you must manually add the appropriate attribute-based format rules to your EDD.

For example, assume you have these declarations:

```
<!ELEMENT par - - (#PCDATA)>
<!ATTLIST par
   font (helv | times | courier) #REQUIRED
   weight (b | r) r>
```

The attributes `font` and `weight` indicate information needed to properly format the paragraph. You can use format rules on the `Par` element to set the font family and weight on the basis of these two attributes so you might use this element definition in your EDD:

Element (Container): Par
General rule: <TEXT>
Attribute list
1.Name: Font **Choice** **Required**
Choices: Helv | Times | Courier
1.Name: Weight **Choice** **Optional**
Choices: B | R
Default: R
Text format rules
1. If context is: [Font = “Helv”]
Default font properties
Family: Helvetica
Else, if context is: [Font = “Times”]
Default font properties
Family: Times
Else
Default font properties
Family: Courier
2. If context is: [Weight = “B”]
Default font properties
Weight: Bold
Else
Default font properties
Weight: Regular

In format rules, you can test for specific attribute values but cannot use the value of an attribute to explicitly set a formatting property. For this reason, if the legal attribute values cannot be enumerated in a format rule, you must write a structure API client to pick up the information. For example, if your `par` element has a numeric attribute `size` intended to specify the font size and you do not know what the legal set of font sizes will be, you cannot set the size font property from that attribute value in a format rule. Instead, you must write an API client to set the size on import of a markup document.

FrameMaker automatically handles certain FrameMaker formatting properties associated with tables, graphics, and equations. For information on these properties, see Chapter 18, "Translating Tables," and Chapter 19, "Translating Graphics and Equations."

For information on writing format rules, see Chapter 10, "Text Format Rules for Containers, Tables, and Footnotes." For information on creating a structure API, see the online manual *Structure Import/Export API Programmer's Guide*, included with the FDK.

17 Translating Entities and Processing Instructions

Entities in markup serve a variety of functions in markup documents. These functions can be represented in widely different ways in FrameMaker. FrameMaker does not have a single construct that corresponds with entities in markup. For information on common uses of entities, see “Entities” on page 17.

Processing instructions (PIs) in markup provide a mechanism for invoking system-specific actions from within an XML or SGML document. These actions can be almost anything. By default, FrameMaker interprets a small number of processing instructions. It stores others in the document as markers—you can write an FDK client to process them at any time in the authoring cycle. You can also use a structure API client to trap processing instructions and perform specific actions as the parser encounters them.

In this chapter

This chapter discusses the default translations used by FrameMaker for processing instructions and for entities of various kinds. The chapter also provides general information on how you can modify these translations. Subsequent chapters discuss further modifications appropriate to translating particular FrameMaker constructs such as variables or books. For information on the processing instructions used for conditional text, see Chapter 23, “Translating Conditional Text.”; for those used for books and book components, see Chapter 24, “Processing Multiple Files as Books.” In the outline below, click a topic to go to its page.

How FrameMaker translates entities and processing instructions by default:

- “On export to markup” on page 248
- “On import to FrameMaker” on page 251

Some ways you can change the default translation:

- “Specifying the location of entity declarations” on page 258
- “Renaming entities that become variables” on page 258
- “Translating entity references on import and export” on page 258
- “Translating entities as FrameMaker variables” on page 259
- “Translating SDATA entities as special characters in FrameMaker” on page 260
- “Translating SDATA entities as FrameMaker text insets” on page 262
- “Translating SDATA entities as FrameMaker reference elements” on page 263

- "Translating external text entities as text insets" on page 264
- "Translating internal text entities as text insets" on page 265
- "Changing the structure and formatting of a text inset on import" on page 266
- "Discarding external data entity references" on page 267
- "Translating ISO public entities" on page 267
- "Facilitating entry of special characters that translate as entities" on page 268
- "Creating book components from general entities" on page 268
- "Discarding unknown processing instructions" on page 268
- "Using entities for storing graphics or equations" on page 269

Default translation

With few exceptions, FrameMaker preserves all of the entity structure of a document. When exporting a document to XML or SGML, FrameMaker generates entity references for user variables, graphics, equations, text insets, books, and book components by default. However, while it can read a DTD that uses parameter entities, the software does not mark the replacement text as an entity within the FrameMaker EDD. When you save the resulting EDD as a DTD, the result will not include the parameter entities that were in the original DTD.

FrameMaker creates processing instructions (PIs) for a small set of special cases on export. Similarly, on import it interprets only the same small set of processing instructions. It retains other PIs as markers and translates them back into PIs on export.

FrameMaker preserves text entities that are references to external text files. On import, these entities are translated as text insets to the referenced file. On export, the software writes the entity reference in the XML or SGML instance; if the entity is not defined in the structure application's DTD, the software writes an entity declaration in the internal DTD subset for the exported document instance.

If a referenced text file is an XML or SGML fragment (at least one element, but no SGML declaration or DTD subset), the software interprets its structure according to the current structure application. If the referenced file is text, the software treats the file's text as the content of the element containing the entity reference.

On export to markup

FrameMaker uses entities for text insets, user variables, graphics, equations, books, and book components. If the entities are not declared in the structure application's DTD, FrameMaker generates appropriate entity declarations in the internal DTD subset of the exported document instance.

If the FrameMaker document was made by importing an XML or SGML instance, and that instance had entities declared in the internal DTD subset, FrameMaker will regenerate those entity declarations when exporting the document to markup.

If the user created a FrameMaker document object that must export as an entity, and there is no information the software can use to map that object to an existing entity declaration, the software will generate a declaration in the internal DTD subset of the document instance.

For information on export of variables, graphics, equations, books, or book components, see Chapter 21, "Translating Variables and System Variable Elements," Chapter 19, "Translating Graphics and Equations," or Chapter 24, "Processing Multiple Files as Books."

Generating processing instructions on export

> **XML and SGML:** The XML specification defines the PI closing delimiter as ?>, while in SGML the closing delimiter is >. This section uses the XML specification to illustrate PI syntax.

By default, FrameMaker creates processing instructions under the following circumstances:

- When the exported markup turns out to be invalid, the software generates a <?FM Validation Off?> PI. When importing an XML or SGML instance with this PI in its DTD subset, FrameMaker imports the complete file whether it is valid or not. Note that this PI is only reliable for XML or SGML files that were exported by FrameMaker.
- When exporting a book, the software generates a <?FM book?> PI to identify a book, and an entity reference, &bkc#; to identify each book component, for example, &bkc1; , &ckc2; , and so forth. Each entities begins with a <?FM document *filename*?> PI. For information on export of books and book components, see Chapter 24, "Processing Multiple Files as Books."
- When exporting a document containing non-element markers of types other than DOC PI or DOC Entity Reference, the software generates a <?FM MARKER [*type*] *text*?> PI where *type* is the marker type and *text* is the marker text. For example, if you have an Index marker that is not in a marker element with the text "lionfish, ocellated", then on export, FrameMaker writes <?FM MARKER [Index] lionfish, ocellated?>.
- When exporting a marker of type DOC PI, the software generates a PI with <?*text*?>, where *text* is the marker text. (For information about how FrameMaker creates a DOC PI marker on import, see "Processing instructions" on page 256.)

Exporting special marker types as processing instructions and entity references

FrameMaker uses special marker types to store information on processing instructions, PI entities, and external data entities.

This marker type	Defines
DOC PI	a processing instruction
DOC Entity Reference	an entity reference, a PI entity, or a reference to a PI entity

Where *text* is the marker text, the software exports `DOC PI` markers as processing instructions of this form:

```
<?text?>
```

and for SGML it exports DOC Entity Reference markers as entity references of this form:

```
&text;
```

Note that you cannot use a read/write rule to change the marker type FrameMaker uses to store PI, entity reference, or PI entity information.

Exporting text insets

FrameMaker text insets generally export as external text entities. On import, if the external entity is declared in the internal DTD subset of the document instance, FrameMaker stores information about the entity on the *Entity Declarations* reference page. On export, the software translates a text inset as follows:

If the text inset references	And	FrameMaker exports it as
A text or markup file, or a markup fragment	The structure application's DTD has an entity definition that maps to the referenced file	An entity reference to the entity declaration in the structure application's DTD.
A text or markup file, or a markup fragment	The *Entity Declarations* reference page maps an entity declaration to the referenced file	An entity declaration in the internal DTD subset for the document instance, using the original entity name. It also writes a reference to that entity declaration.
A text or markup file, or a markup fragment	The *Entity Declarations* reference page *does not* map an entity declaration to the referenced file, nor is there a corresponding entity declaration in the structure application's DTD	An entity declaration in the internal DTD subset for the document instance, using a generated entity name. It also writes an entity reference to that entity declaration.
A FrameMaker text flow or any text inset not mentioned above		Markup and content in the document instance—the software does not export an entity or entity reference.

On import to FrameMaker

By default, FrameMaker interprets entity references as appropriate document objects, and saves enough information to recreate the associated entity declarations on export. The software stores information about the entity declaration in the following ways:

- The entity name is stored with the FrameMaker document object that corresponds with the original entity reference (as it was in the XML or SGM).
- If the entity was declared in the internal DTD subset of the document instance, the software stores that information on the Entity Declarations reference page of the resulting FrameMaker document.
- If the entity was declared in the external DTD subset, the software assumes it will be declared in your structure application's DTD. No entity declaration information is stored with the FrameMaker document.

In this way, the software can import and export entity references correctly. You can also modify your DTD or use read/write rules to have FrameMaker treat some entities differently.

The rest of this section describes how FrameMaker imports various entities and unknown processing instructions in the absence of rules or clients.

Internal text entities

XML: The XML specification doesn't treat markup data as literals in an internal text entity. For example, a reference to the following entity:

```
<!ENTITY example "<p>An ampersand (&#38;#38;) may be escaped
numerically (&#38;#38;#38;) or with a general entity
(&amp;amp;).</p>" >
```

appears in a document as:

An ampersand (&) may be escaped numerically (&) or with a general entity (&).</p>

Internal XML or SGML text entities are text entities whose replacement text is determined solely by information in their declarations. For SGML, the parser recognizes any markup within the entity text as such. The following are examples of declarations of internal text entities—The first example is valid in XML and SGML, while the second and third examples are only valid in SGML:

```
<!ENTITY product "FrameMaker">
```

```
<!ENTITY blist "<list type=bulleted>">
```

```
<!ENTITY blist STARTTAG "list type=bulleted">
```

On import, FrameMaker tests the entity text to make sure it has no characters which can be recognized as markup in any context (within the SGML reference delimiter set).

If the entity text has no markup characters (as in the first example), FrameMaker translates it to a variable of the same name. If the document template does not define a variable of that name, FrameMaker uses the entity text as the variable's definition.

When importing SGML, if the template already contains a variable with that name, FrameMaker uses the template's variable definition. It is possible that this match between entity and variable names is unintentional, but FrameMaker does not write a message to the log file. The file imports without complaining and FrameMaker uses the template's definition for the variable. However, in all cases the information on the *Entity Declarations* reference page contains the full text from the original entity declaration.

When importing XML, FrameMaker discards any definition of the variable in the template and creates a new one from the XML document's entity declaration.

For information on structure log files, see "Log files" on page 71.

SGML: The default maximum length is 240 characters when no `sgmldcl` file is included in your `sgmlapplication` file. If the replacement text is greater than 240 characters, FrameMaker imports the entity as plain text. Messages are written to the Error Log stating, "The Normalized length or literal exceeded 240; markup terminated." and "Length of name, number, or token exceeded the NAMELEN limit". If the entity text contains any markup characters (as in the last two examples above), FrameMaker imports the entity as plain text and discards all information about the entity reference.

Internal character data (CDATA) entities

XML: The XML specification does not support CDATA entities—the following information pertains to SGML, only.

Internal `CDATA` entities are similar to internal SGML text entities, except that the parser treats the entity text as character data only. That is, the parser does not recognize any markup within the entity text. The following is an internal `CDATA` entity:

```
<!ENTITY tag CDATA "<TAG>">
```

FrameMaker always translates these entities as variables. In the above example, the resulting variable text would be `<TAG>`.

Internal special character data (SDATA) entities

XML: The XML specification does not support `SDATA` entities—the following information pertains to SGML, only. However, you can declare text entities that FrameMaker will convert to variables as follows:

```
<!ENTITY date "FM variable: Current Date (Long)">
```

Internal SDATA entities carry specific information for the system to interpret when processing the document. You can use such entities to represent special characters or canned text. The following are examples of internal SDATA entities:

```
<!ENTITY bullet SDATA "[bullet]">

<!ENTITY copyright SDATA "[Copyrt]">

<!ENTITY date SDATA "FM variable: Current Date (Long)">
```

In the first example, the intent is to have the system interpret the entity as the bullet symbol character. In the second example, the intent is to insert canned text representing the copyright notice for the document. These two declarations are not specific to a particular system. On the other hand, the third declaration is specific to FrameMaker. In this case, the intent is to use the FrameMaker system variable Current Date (Long).

Since FrameMaker has access to the DTD containing the entity declaration, you can control the import and export of an SDATA entity by supplying an appropriate parameter literal in its entity declaration in the DTD. If you do so, you do not need to use read/write rules for this purpose. FrameMaker has a convention for interpreting certain parameter literals containing text that starts with one of these character sequences:

A literal beginning with:	Interprets the SDATA entity as:
`fm char:`	the specified FrameMaker character
`fm ref:`	the specified reference element
`fm text inset:`	a text inset importing the specified file
`fm variable:`	the named variable

For example, to translate the SDATA entity `oquote` to the directional open quotation mark (“) in FrameMaker, you could use this declaration in your DTD:

```
<!ENTITY oquote SDATA "FM char: \xd2">
```

With this entity declaration, when FrameMaker encounters a reference to the `oquote` entity as it is importing an SGML document, it replaces the reference with a directional open quotation mark. When it encounters a directional open quotation mark as it is exporting a FrameMaker document, it generates an `oquote` entity reference. In either case, you do not need rules to accomplish the translation. For information on how FrameMaker interprets each of these parameter literals, see the procedures in “Modifications to the default translation” on page 257.

If FrameMaker does not recognize the parameter literal, it translates the entity as a variable of the same name, with the variable text determined as for internal text entities. If the template already contains a variable with that name, FrameMaker uses the template’s variable definition.

If FrameMaker automatically creates the variable definition during import, it wraps the variable text in the FmSdata character format. This character format distinguishes SDATA entity text from regular text and tells FrameMaker what type of entity to create on export.

External data entities

XML: This section discusses the use of external entities for graphic images and other types of non-SGML data. The XML specification refers to this type of entity as a nonparsed entity. According to the XML specification, any nonparsed entity must be referenced via an `entity` or `entities` attribute of an element.

External data entities, unlike any of the other types of entities, are associated with a `notation` name. A `notation` is a mechanism by which the system recognizes how to process the data referenced by the entity. While external data entities may be used for any number of purposes, they are typically used to reference text, graphics, equations, and tables residing in external files. The following is an example of the declarations needed for an external data entity that specifies a graphic file:

```
<!NOTATION cgm SYSTEM "cgm2mif">
<!ENTITY door SYSTEM "door.cgm" NDATA cgm>
```

SGML: The above example uses XML syntax. For SGML the notation would exclude quotes for the notation identifier as follows: `<!NOTATION cgm SYSTEM cgm2mif>`

Except for external text entities (see "External text entities," next), FrameMaker translates a direct reference to an external entity as a marker of type `DOC Entity Reference`. For this reason, we suggest graphic elements use an `entity` attribute to specify the entity reference.

For example, a reference to the above entity should be made in a graphic element's `entity` attribute. Given appropriate read/write rules to import the element as a FrameMaker graphic element, the software would create an anchored frame that imports `door.cgm` by reference. For more about importing graphic elements, see Chapter 19, "Translating Graphics and Equations."

XML: This use of an `entity` attribute to specify the entity reference is the only method allowed by the XML specification to reference a nonparsed entity (such as a graphic or a sound file).

If the entity declaration was made in the XML or SGML document's internal DTD subset, the software saves the entity information on the *Entity Declarations* reference page of the resulting FrameMaker document. This will be used to reconstruct the entity declaration on export.

You can use a read/write rule to drop references to all external data entities on import; see "external data entity reference" on page 391.

If you reference an external data entity as the value of an attribute, you may want to write a rule to import it as a FrameMaker graphic. For more information about using entity references in attributes to import graphic files, see Chapter 19, "Translating Graphics and Equations."

External text entities

External text entities specify an external file which can contain either markup or character data. In effect, the contents of the external file replaces the entity reference and the parser recognizes any markup within the replacement text.

As examples:

```
<!ENTITY appendix SYSTEM "appendix.sgm">

<!ENTITY cpyright SYSTEM "cpyrt.txt">

<!ENTITY cpyright PUBLIC "//. . . public-id . . .//"
       "/a/corp/cpyrt.txt">
```

FrameMaker expands these entities and imports their replacement text (including any structure) as a text inset. If the entity declaration is specified in the internal DTD subset of the document instance, the software saves information about the entity declaration on the *Entity Declarations* reference page. This will be used to reconstruct the entity declaration on export.

In some cases, FrameMaker translates some external text entities as book components. On export, it translates these text insets and book components back into external text entities. For more information on working with books, see Chapter 24, “Processing Multiple Files as Books.”

Parameter entities

Parameter entities can be referenced only within declarations. Typically they are used for such things as defining common segments of content models or common attribute definition lists. The following are examples of parameter entities:

```
<!ENTITY % subelts "(quote | emphasis | code | acronym)">

<!ENTITY % comnatt "id ID #IMPLIED
    type CDATA #IMPLIED
    security (ts, c, uc) uc">
```

FrameMaker expands these entities and imports their replacement text. On export, FrameMaker exports only the entity replacement text.

Subdocument (SUBDOC) entities

> **XML:** The XML specification does not support SUBDOC entities—the following information pertains to SGML, only.

SUBDOC entities are inclusions of a complete document within another. The master document and its subdocuments each have their own document type definitions and are validated accordingly. FrameMaker does not support SUBDOC entities. If you try to open an SGML document that uses SUBDOC entities, the software reports an error in the log file and does not open the file.

PI entities

> **XML:** The XML specification does not support PI entities—the following information pertains to SGML, only.

PI entities are processing instructions in the form of entities. They are a convenient way of providing one level of indirection, allowing the user to change the processing instructions in one place if the document is moved to a different system. In addition, using a PI entity allows the processing instruction to contain the `PIC` delimiter (> in the reference concrete syntax). The following is an example of a PI entity:

```
<!ENTITY break PI "MYSYS: pgbrk">
```

Unless PI entities correspond to one of the forms supported by FrameMaker (to represent books and book components), FrameMaker stores them in markers of type `SGML Entity Reference` by default. In the above example, the marker text would be:

```
break
```

Processing instructions

> **XML and SGML:** The XML specification defines the PI closing delimiter as ?>, while in SGML the closing delimiter is >. This section uses the XML specification to illustrate PI syntax.

As stated at the beginning of this chapter, processing instructions in XML or SGML provide a way to perform system-specific actions on a markup document. By default, FrameMaker recognizes a small set of processing instructions. Those it does not recognize in a document instance it stores in a marker of type `DOC PI`. (You can change the marker type used for this purpose with a rule.) For example, if your document instance contains this processing instruction:

```
<?mypi?>
```

then FrameMaker creates a `DOC PI` marker with this marker text:

```
mypi
```

In addition to processing instructions for books, book components, and conditional text, FrameMaker recognizes another processing instruction format it uses to create non-element markers. For example, if your document instance contains this processing instruction:

```
<?FM MARKER [MyMarkerType] Some marker text here?>
```

then FrameMaker creates a `MyMarkerType` marker with this marker text:

```
Some marker text here
```

You can use a read/write rule to drop processing instructions on import or to specify a different marker type to store this information.

Limitations to Writing Processing Instructions

An XML or SGML document should not have processing instructions after the end tag for the highest-level element in the document. For example, even though it is valid, the following document generates errors when you open it in FrameMaker:

```
<!DOCTYPE example [
<!ELEMENT example   ANY>
<!ELEMENT par    (#PCDATA)>
]>

<example>
<par>This is a paragraph.</par>
<?pi between pars?>
<par>Another paragraph.</par>
</example>
<?pi at end?>
```

FrameMaker aborts the import process when it encounters the `<?pi at end>` processing instruction. Because the document is complete when FrameMaker aborts the parsing, the result is a fully imported FrameMaker document. However, errors of this sort could affect structure import/export clients that rely on parsing events that occur after the document is complete.

Modifications to the default translation

> **XML:** The XML specification doesn't allow SDATA or CDATA entities. Many of the following sections discuss modifications to default handling of these types of entities, and so they pertain to SGML, only.

You can handle internal SDATA entities in a variety of ways in FrameMaker, depending on the information they represent. In some cases, you can convert an SDATA entity to a FrameMaker variable or to particular characters using selected character formats. In other situations, you might convert the entity to one or more FrameMaker objects. For example, you may have an entity for a company's logo, which you represent in FrameMaker with an anchored frame.

As will be described in the following sections, you can modify the treatment of some entities either by changing the entity declaration in the DTD or by adding a rule to your read/write rules document. These modifications affect every instance of a reference to the specific entity. With read/write rules, you can either drop an entity, import it as a text inset, or import it as a specified non-element variable. You can also map the entity to an element that is stored on the SGML Utilities reference page of the resulting FrameMaker document.

You can declare SDATA entities with FrameMaker parameter literal text. A parameter literal is a way to declare in the entity itself how FrameMaker will translate the entity. If, for a single entity, you have both an entity declaration with a FrameMaker parameter literal and a rule

that applies to the entity, the entity declaration takes precedence. For an example of parameter literal text, see "Translating entities as FrameMaker variables" on page 259.

For a summary of read/write rules relevant to translating entities, see "Entities" on page 357.

Specifying the location of entity declarations

Your structure application may include files of entity declarations to use in addition to declarations in a particular XML or SGML document. You need to tell FrameMaker where it can find these declarations. You do so in the application definition. For more information, see "Application definition file" on page 50.

Renaming entities that become variables

FrameMaker translates many entities, by default, as variables. You may choose to change the name of the variable or entity. To do so, you use the following rule:

```
entity "ename" is fm variable "var";
```

where *ename* is an entity name in markup, and *var* is a FrameMaker variable. For example, if you have an entity `date`, the FrameMaker variable's name is by default `Date`. To change this behavior to use a system variable you could use this rule:

```
entity "date" is fm variable "Modification Date (Short)";
```

For information on these rules, see "entity" on page 381 and "is fm variable" on page 441.

Translating entity references on import and export

> **XML and SGML:** The entities discussed in this section are typically declared as SDATA entities in SGML. However, SDATA entities are not allowed in XML. The following sections discuss ways to make SDATA entities in SGML correspond with FrameMaker document objects. When working with XML, the preferred method is to use specific elements and map them to the document objects.

When working with XML or SGML, you can use read/write rules to convert an entity into a FrameMaker document object such as a variable or a reference element. With SGML, you can also specify a FrameMaker *parameter literal* in the entity declaration of the DTD—the parameter literal declares that FrameMaker will convert the entity into the desired document object. You must keep in mind that his approach introduces application-specific data into the DTD.

FrameMaker parameter literals and read/write rules have the same effect, but with read/write rules you keep the application-specific outside of the DTD. The parameter literals and corresponding uses of the `entity` rule are described in the following sections. In brief, they are:

For	Parameter Literal	Rule	Page
Special characters	`"fm char:"`	`entity "ename" is fm char`	260

For	Parameter Literal	Rule	Page
Variables	`"fm variable:"`	`entity "ename" is fm variable`	259
Text insets	`"fm text inset:"`	`entity "ename" is fm text inset "fname"`	262
Other entities	`"fm ref:"`	`entity "ename" is fm reference element`	263

If you use the `entity` rule, you can choose to have it effective only when the software reads a markup document, not when it writes a FrameMaker document as XML or SGML. To do so, you make the `entity` rule a subrule of a highest-level `reader` rule. For example, an SGML document might use a `period` entity for some instances of the period (.) character. If you use this rule:

```
entity "period" is fm char ".";
```

then on export, all periods in the document become references to the `period` entity. To have periods remain the period character on export, use this version of the rule instead:

```
reader
   entity "period" is fm char ".";
```

For information on these rules, see "entity" on page 381, "is fm char" on page 416, and "reader" on page 453.

Translating entities as FrameMaker variables

> **XML:** The XML specification doesn't allow SDATA entities. To translate FrameMaker system variables in XML, you should translate them to specific XML elements. You can similarly translate user variables, or translate them as internal text entities.

You can equate an entity with a FrameMaker variable for import and export. You can do this by using either the appropriate version of the `entity` rule or the parameter literal. The parameter literal in the entity declaration has the form:

```
"FM variable: var"
```

The `entity` rule has the form:

```
entity "ename" is fm variable "var";
```

where *ename* is the entity in markup, and *var* is the desired FrameMaker variable.

For example, to have an entity named `date` correspond to the FrameMaker system variable Current Date (Long), you can add this entity declaration to your DTD:

```
<!ENTITY date SDATA "FM variable: Current Date (Long)">
```

Alternatively, if your DTD contains the entity declaration:

```
<!ENTITY date SDATA "[date]">
```

You could leave the DTD as is and add the following rule to your rules document:

```
entity "date" is fm variable "Current Date (Long)";
```

Important: If you import an SGML document that inserts a FrameMaker system variable into the document, you must save the file and update variables in the document before system variables such as Creation Date (Long) appear. For information on updating variables, see the information about variables in the using manual for FrameMaker.

You can choose to have this translation only happen on import of an SGML document. For more information, see "Translating entity references on import and export," (the previous section).

For information on these rules, see "entity" on page 381 and "is fm variable" on page 441. For more information on the treatment of FrameMaker variables, see Chapter 21, "Translating Variables and System Variable Elements."

Translating SDATA entities as special characters in FrameMaker

XML: The XML specification doesn't allow SDATA entities. However, you can translate special characters as internal text entities.

A common usage of SDATA entities is to enter special characters. In this situation, you could translate the SDATA entity in your SGML document directly as the appropriate character in FrameMaker, through either the appropriate version of the `entity` rule or the parameter literal.

The parameter literal in the entity declaration has the following form:

```
"fm char: code [in fmchartag]"
```

The `entity` rule has one of these forms:

```
entity "ename" is fm char "char" [in "fmchartag"];
```

```
entity "ename" is fm char code [in "fmchartag"];
```

where *ename* is an SGML entity, *code* is a character code (specified as either a 1-character string or an integer constant, using the syntax for an octal, hexadecimal, or decimal integer described in "Strings and constants" on page 212). Note that if the desired character is a digit or a white-space character, you must enter it as an integer. *char* is a one-character string and *fmchartag* is an optional FrameMaker character tag.

Without the `in` clause, this parameter literal or rule causes FrameMaker to simply insert the character specified by *`code`* or *`char`* when importing an SGML document.

> **Important:** Special characters often require a font change. For either the rule or the parameter literal, the `in` clause tells FrameMaker to insert the indicated character using the specified character format. Note that the character format must specify a non-standard font such as Symbol or Zapf Dingbats. Otherwise, this clause cannot override the formatting for the parent element.

For example, to have an entity named `cquote` corresponding to the directional close quotation mark (”), you can add this entity declaration to your DTD:

```
<!ENTITY cquote SDATA "FM char: \xd3">
```

Alternatively, if your DTD contains an entity declaration such as:

```
<!ENTITY cquote SDATA "close-quote">
```

you can add one of the following rules to your rules document:

```
entity "cquote" is fm char "\”";
```

```
entity "cquote" is fm char "\xd3";
```

That is, your rule can specify a string with the special character or it can specify the numeric character code. The numeric code can be octal, hexadecimal, or decimal; `0xd3`, `\xd3`, and `211` all represent the same close-quote character.

For example, create a character format named Trademark, specify Symbol as the font family, and turn on the superscript property. To translate the `SDATA` entity `reg` as a superscripted version of the registered trademark character (®), you could use this declaration in your DTD:

```
<!ENTITY reg SDATA "FM char: 0xd2 in Trademark">
```

or have this rule in your rules document:

```
entity "reg" is fm char 0xd2 in "Trademark";
```

When FrameMaker encounters a reference to the `reg` entity when importing an SGML document, it replaces the reference with ® (assuming your FrameMaker template defines the Trademark character format appropriately). When exporting a document, if FrameMaker encounters ® in the Trademark character format, it generates a reference to the `reg` entity.

If you translate entities as special characters, you may want to create entity palettes to make it easier for your end users to put these special characters in a FrameMaker document. For information on how to create entity palettes, see “Facilitating entry of special characters that translate as entities” on page 268.

DTDs frequently use the entity sets defined in Annex D of the SGML Standard, often called ISO public entity sets, for providing commonly used special characters. FrameMaker includes copies of these entity sets and provides rules to handle them for your application.

For information on how FrameMaker supports ISO public entities, see Appendix F, "ISO Public Entities."

You can choose to have this translation only happen on import of an SGML document. For more information, see "Translating entity references on import and export" on page 258.

For information on the rules used in these examples, see "entity" on page 381 and "is fm char" on page 416. For information on the FrameMaker character set, see Appendix E, "Character Set Mapping," in this manual, and see the FrameMaker user's manual.

Translating SDATA entities as FrameMaker text insets

> **XML:** The XML specification doesn't allow SDATA entities. However, FrameMaker translates external text entities as text insets by default. See "Translating external text entities as text insets" on page 264.

You may want to translate some SGML SDATA entities to text insets in the resulting document. Note that the file you use for such a text inset must be valid SDATA. In other words, the source file must not be an SGML file.

The source file can be of any document type that FrameMaker can filter automatically. Typically, such files will be FrameMaker documents or text files, although you can use files created by other word processing systems. If the source file is a FrameMaker document, a text inset is always the entire contents of a flow.

If the source of the text inset is a structured flow, the document that is importing it will not be valid if the flow has a highest-level element. There is one exception however, for structured flows that have `SGMLFragment` as the highest-level element. An SGML fragment is an SGML instance that contains neither an SGML declaration nor a DTD subset. Opening an SGML fragment in FrameMaker generates a structured document with a highest-level element named `SGMLFragment`. When you import such a structured flow as a text inset, FrameMaker automatically unwraps the `SGMLFragment` element so it's children will be valid in the document.

You can translate an SDATA entity to a text inset either through the appropriate version of the `entity` rule or the parameter literal.

The parameter literal in the entity declaration has one of the following forms:

```
"FM text inset: fname"
"FM text inset: fname in body flow flowname"
"FM text inset: fname in reference flow flowname"
```

The `entity` rule has one of these forms:

```
entity "ename" is fm text inset "fname";

entity "ename" is fm text inset "fname"
   in body flow "flowname";
```

```
entity "ename" is fm text inset "fname"
   in reference flow "flowname";
```

where *ename* is an SGML entity, *fname* is the document that is the source of the text inset, and *flowname* is a flow in fname. The file named by *fname* does not have to be a FrameMaker document. If it is, you can specify the flow to use. If you do not specify a flow, the main body flow of the document is used.

For example, if you have a copyright notice on the main flow of a document named `copy.fm`, you can use this entity declaration:

```
<!ENTITY copyrt SDATA
   "FM text inset: copy.fm in body flow A">
```

Alternatively, you can use this rule:

```
entity "copyrt" is fm text inset "copy.fm" in body flow "A";
```

Insertion of a FrameMaker text inset in a document always inserts an end of paragraph in the document. For this reason, you should only use this rule for entities that translate to entire paragraphs or that occur only at the end of a paragraph.

By default, the structure of a text inset is retained and the text reformatted to match the FrameMaker document into which it is placed. You may choose to change this behavior. For information on how to do so, see "Changing the structure and formatting of a text inset on import" on page 266.

You can choose to have this translation only happen on import of an SGML document. For more information, see "Translating entity references on import and export" on page 258.

For information on these rules, see "entity" on page 381 and "is fm text inset" on page 437.

Translating SDATA entities as FrameMaker reference elements

> **XML:** The XML specification doesn't allow SDATA entities. This section pertains to SGML, only.

In addition to the common situations discussed in the previous sections, you may use an SDATA entity for almost any set of objects in your document. For example, an entity can be used to hold one or more anchored frames. To handle some of these cases, you can create a correspondence between an SDATA entity and an element on a *reference page* in the FrameMaker template associated with your application. To do so, use either the appropriate version of the `entity` rule or the parameter literal.

The parameter literal in the entity declaration or the SDATA rule has the form:

```
"fm ref: fmtag"
```

The `entity` rule has the form:

```
entity "ename" is fm reference element "fmtag";
```

where *fmtag* is the name of an element on a FrameMaker reference page in the associated FrameMaker template and *ename* is an SGML entity.

The *fmtag* element must occur in a flow named `Reference Elements`. That flow must be on a reference page with a name that starts with `SGML Utilities Page`—for example, `SGML Utilities Page 1` or `SGML Utilities Page Logos`. For information on working with reference pages, see the FrameMaker using manual.

When FrameMaker encounters references to the specified entity while importing an SGML document, it copies the appropriate element from its reference page in the associated FrameMaker template. When it encounters an instance of an element associated with one of the reference pages while exporting a document, it generates an entity reference.

For example, to have an entity named `logo` correspond to an anchored frame with your company's logo, you can add this entity declaration to your DTD:

```
<!ENTITY logo SDATA "fm ref: Logo">
```

Alternatively, if your DTD contains an entity declaration such as:

```
<!ENTITY logo SDATA "[logo]">
```

you can add the following rule to your rules document:

```
entity "logo" is fm reference element "Logo";
```

You can choose to have this translation only happen on import of an SGML document. For more information, see "Translating entity references on import and export" on page 258.

For information on these rules, see "entity" on page 381 and "is fm reference element" on page 430.

Translating external text entities as text insets

External text entities specify a reference to an external file that is either markup or text. By default, on import FrameMaker translates the entity as a text inset. Note that the specified file cannot be a FrameMaker document because such an entity declaration would not be valid markup.

If the referenced entity declaration was made in the DTD subset of the markup document, FrameMaker stores information about it on the Entity Declarations reference page. On export, it uses that information to generate entity declarations in the DTD subset of the markup.

Unless the source file is markup, insertion of a FrameMaker text inset in a document always inserts an end of paragraph in the document. For this reason, you should be careful about which external text entities must translate to entire paragraphs or occur only at the end of a paragraph.

FrameMaker has options to update text insets manually or automatically whenever the user opens the document containing the text insets. By default, external text entities import as

text insets that update manually. Authors can select these text insets to change this setting through the user interface.

An author can manually import markup fragments into a document as text insets. When exporting to markup, FrameMaker might not have entity declarations for such user-created text insets. In the absence of entity declarations, FrameMaker generates entity declarations with entity names of `ti1`, `ti2`, etc.

Translating internal text entities as text insets

By default, on import FrameMaker translates internal text entities either as variables or as plain text. You can choose to have it instead import a text entity as a FrameMaker text inset. You may do so if the FrameMaker cannot import the entity as a variable. For example, the entity might contain too many characters, or (for SGML) it might contain SGML elements.

To import a text entity as a FrameMaker text inset, use one of these forms of the `is fm text inset` rule:

```
entity "ename" is fm text inset fname;

entity "ename"
   is fm text inset "fname" in body flow "flowname";

entity "ename"
   is fm text inset "fname" in reference flow "flowname";
```

where *ename* is the entity name, *fname* is a file path to a FrameMaker document, and *flowname* is a flow in the FrameMaker document. This rule translates the entity *ename* as a text inset. The inset is a reference to the flow named *flowname* in the document named *fname*.

For example, assume you have this entity declaration:

```
<!ENTITY copyrt
   "COPYRIGHT (c) 1500-1995 OLD GUYS CORPORATION
   The source code contained herein is our very own.
   You can't use it unless we say so.">
```

To import `copyrt` as a text inset that references a flow named Copyright on the reference page of an existing FrameMaker document named copyrt.fm, use this rule:

```
entity "copyrt"
   is fm text inset "copyrt.fm" in reference flow "Copyright";
```

The source file can be of any document type that FrameMaker can filter automatically. Typically, such files will be FrameMaker documents or text files, although you can use files created by other word processing systems. Also, a FrameMaker text inset is always the entire contents of a flow or file.

Insertion of a FrameMaker text inset in a document always inserts an end of paragraph in the document. For this reason, you should use this rule for entities that translate to entire paragraphs or that occur only at the end of a paragraph.

If the source of the text inset is a structured flow, the document that is importing it will not be valid if the flow has a highest-level element. There is one exception however, for structured flows that have `SGMLFragment` as the highest-level element. An SGML fragment is an SGML instance that contains neither an SGML declaration nor a DTD subset. Opening an SGML fragment in FrameMaker generates a structured document with a highest-level element named `SGMLFragment`. When you import such a structured flow as a text inset, FrameMaker automatically unwraps the `SGMLFragment` element so it's children will be valid in the document.

By default, the structure of a text inset is retained and the text reformatted to match the FrameMaker document into which it is placed. You may choose to change this behavior. For information on how to do so, see "Changing the structure and formatting of a text inset on import," next.

You can choose to have this translation only happen on import of a markup document. For more information, see "Translating entity references on import and export" on page 258.

For information on these rules, see "entity" on page 381 and "is fm text inset" on page 437.

Changing the structure and formatting of a text inset on import

As described in "Translating SDATA entities as FrameMaker text insets" on page 262 and "Translating internal text entities as text insets," (the previous section), you can choose to import some entities as FrameMaker text insets. When you do so, the inset is to a FrameMaker document file. The source document's structure (if any) is retained in the resulting text inset by default. The text is also reformatted according to the format rules of the target document. You may wish to change this behavior. To do so, you use one of the following rules:

```
entity "ename" {
   is fm text inset "fname";
   reformat using target document catalogs;
}

entity "ename" {
   is fm text inset "fname";
   reformat as plain text;
}

entity "ename" {
   is fm text inset "fname";
   retain source document formatting;
}
```

where *ename* is the entity name and the `is fm text inset` rule has the arguments described in "Translating internal text entities as text insets," (the previous section).

The meaning of these rules is as follows:

- `reformat using target document catalogs`: Retain the structure of the source document and use the formatting contained in the target document. This is the default behavior.
- `reformat as plain text`: Remove the structure of the source and use the formatting contained in the target document.
- `retain source document formatting`: Remove the structure of the source and use the formatting of the source text.

In a single `entity` rule, you can use one of these rules at most. Also, you cannot use one of these rules as a subrule of an `entity` rule unless that `entity` rule also has an `is fm text inset` subrule.

You can use any of these rules at highest level, to set the default treatment of all text insets. If the source file is text or markup, these rules will have no effect. For example, for insets to FrameMaker documents, if you always want the structure stripped from the inset and you want the text to retain the formatting of the original document, you use this rule:

```
retain source document formatting;
```

For more information on these rules, see "entity" on page 381, "is fm text inset" on page 437, "reformat as plain text" on page 454, "reformat using target document catalogs" on page 455, and "retain source document formatting" on page 456.

Discarding external data entity references

By default, FrameMaker imports direct references to external data entities as markers of type `DOC Entity Reference`. Instead, you can choose to have it discard these references. To do so, use the following highest-level rule:

```
external data entity reference drop;
```

In SGML, the values of general entity name attributes, such as those used with graphics, are not considered entity references. This rule does not affect how FrameMaker treats general entity name attributes. For example, if a graphic element specifies an entity named `mygraphic` in its `entity` attribute, the entity `mygraphic` will not be affected.

For more information on these rules, see "external data entity reference" on page 391 and "drop" on page 373.

Translating ISO public entities

For information on how FrameMaker supports ISO public entities, see Appendix F, "ISO Public Entities."

Facilitating entry of special characters that translate as entities

> **XML:** The XML specification allows UNICODE in XML. This section is for SGML, only.

Your end users do not need to know the details of the entity names or definitions used to represent special characters within a FrameMaker document. They can simply insert any special character into the document and have FrameMaker automatically translate the characters back and forth to the corresponding entity references.

However, you may want to provide a special interface for inserting these characters. You can create a hypertext palette for this purpose.

You build a hypertext palette for inserting special characters with whatever layout you want. Below each active region representing a special character within the palette, you store a hypertext command to have FrameMaker insert the appropriate entity. This hypertext command has the following form:

```
message FmDispatcher insert entity ename
```

where *ename* is the entity name. When your end user clicks on a particular character within the palette, the hypertext message is sent to FrameMaker requesting the insertion of an entity with the given name. FrameMaker determines what to insert on the basis of current read/write rules and the entity declaration. This could result in inserting, for example, a special character, an SDATA variable, or a text inset.

Note that FrameMaker can insert the document objects you specify for these entities, whether or not there is a matching entity declared in the DTD. You should be sure to declare a matching entity for each document object. Without an entity declaration, on export the software cannot convert the object into a DOC Entity Reference.

For more on creating hypertext documents, see the information about hypertext documents in the *FrameMaker User Guide*.

Creating book components from general entities

You can break large documents across multiple files and manage those files with a single document containing general entity references in markup, and with a book file in FrameMaker. For information on this use of general entities, see Chapter 24, “Processing Multiple Files as Books.”

Discarding unknown processing instructions

By default, FrameMaker imports processing instructions it does not understand as markers of type `DOC PI`. (In previous versions of FrameMaker, these were Type 11 markers.) You can choose to have it discard these processing instructions instead. To do so, use the following highest-level rule:

```
processing instruction drop;
```

For more information on these rules, see “processing instruction” on page 451 and “drop” on page 373.

Using entities for storing graphics or equations

In markup, you often store graphics and equations in separate files and then include them in the document with general entity name attributes. For information on this use of general entities, see Chapter 19, “Translating Graphics and Equations.”

18 Translating Tables

Many documents require the use of tables to organize information into cells in rows and columns. FrameMaker has a complete tool for creating tables and has a specialized element structure for representing them. Markup does not standardize the representation of tables; each DTD can represent tables differently. In practice however, many DTDs use the element and attribute declarations for tables developed for the CALS initiative, which FrameMaker directly supports.

DTDs that do not use the CALS table model can have arbitrary representations for tables. To support arbitrary table models, FrameMaker needs you to provide information in the form of read/write rules.

FrameMaker supports the CALS table model in the sense that you can import XML and SGML documents that use the CALS table model without providing read/write rules for the translation. The software automatically recognizes these elements and attributes and creates corresponding FrameMaker tables. In some situations, of course, you may choose to modify how the software creates these tables.

In this chapter

This chapter discusses how FrameMaker interprets the CALS table model: what it does by default when reading a markup document that uses CALS tables and how you can change that with read/write rules. It also discusses the default handling of arbitrary tables and how you can change that handling. In the outline below, click a topic to go to its page.

How FrameMaker translates tables by default:

- "On import to FrameMaker" on page 272
- "On export to markup" on page 275

Some ways you can change the default translation:

- "Formatting properties for tables" on page 276
- "Identifying and renaming table parts" on page 279
- "Representing FrameMaker table properties as attributes in markup" on page 280
- "Representing FrameMaker table properties implicitly in markup" on page 281
- "Adding format rules that use CALS attributes (CALS only)" on page 282
- "Working with colspecs and spanspecs (CALS only)" on page 283
- "Specifying which part of a table a row or cell occurs in" on page 283

- "Specifying which column a table cell occurs in" on page 284
- "Omitting explicit representation of table parts" on page 285
- "Creating parts of a table even when those parts have no content" on page 287
- "Specifying the ruling style for a table" on page 289
- "Exporting table widths proportionally" on page 290
- "Creating vertical straddles" on page 290
- "Using a table to format an element as a boxed set of paragraphs" on page 293
- "Creating tables inside other tables" on page 295
- "Rotating tables on the page" on page 295

Default translation

FrameMaker represents tables and table parts as elements with substructure. The CALS model also represents tables and table parts as elements with substructure. The two models are analogous, but have several differences. You need to understand these models in themselves before you can understand how the software translates between them. If you need help with these basics, read the information described in the following paragraph before going on with the sections on import and export that follow it.

For a description of how you use FrameMaker elements to represent tables, see Chapter 9, "Structure Rules for Containers, Tables, and Footnotes." For a description of the element and attribute structure of the CALS table model, see Appendix B, "The CALS Table Model."

On import to FrameMaker

If your DTD does not use the CALS table model and you create an EDD from the DTD or read a markup document into FrameMaker, the software cannot identify elements that correspond to tables and table parts. In this case, you need to write read/write rules to set up the correspondence.

However, if the DTD uses the CALS table fragment, the software creates container and table part elements corresponding to these declarations. The software retains some of the attributes in the markup as attributes in FrameMaker. For other CALS attributes the software uses their values to format the table when reading markup data. The following sections provide more details of the default translation of CALS tables.

You can think of the software's default behavior In translating CALS tables as though it had a built-in set of read/write rules that identify the element and attribute structure of those tables. Appendix C, "Read/Write Rules for CALS Table Model," describes the rules that could be used to mimic this default behavior.

How CALS elements translate

If you create an EDD from a DTD that contains the CALS table declarations, the software translates the CALS `table` element to a container element and translates the `tgroup` element to a FrameMaker table element. Other elements, such as `thead` and `entry`, become table part elements of the appropriate type.

The content model for a CALS `tgroup` requires `thead` and `tfoot` to precede `tbody`. However, a FrameMaker table requires the table body to precede the table footing. When importing a `tgroup` element definition, FrameMaker can switch the order of the `tbody` and `tfoot` elements to match the order required for its tables, but only if the content model for a Tgroup is the following:

```
<!ELEMENT tgroup - O (colspec*, spanspec*, thead?, tfoot?, tbody)>
```

For any other `tgroup` content model, FrameMaker might try to switch the order of the `tbody` and `tfoot` elements upon import. However, because a `tgroup` content model can be arbitrarily complex, we cannot guarantee a valid result. For that reason, the software displays a warning message that says the content model for your table might be incorrect.

> **Important:** In a FrameMaker table, the table heading must precede the table body, and the table body must precede the table footing. If the resulting Table content model in your EDD does not specify the correct order for table parts, you must modify the EDD. Otherwise, tables will not import into your FrameMaker document.

Also, FrameMaker does not create elements corresponding to the `colspec` and `spanspec` elements. These elements exist only for their attributes. When you open a markup document, the software uses the attribute values for `colspec` and `spanspec` elements to create the corresponding table. For more information, see "How colspec and spanspec elements translate" on page 274.

If you have this SGML element structure in a DTD:

```
<!ELEMENT table - - (title?, tgroup+)>
<!ELEMENT tgroup - O (colspec*, spanspec*, thead?, tfoot?, tbody)>
```

the software creates this FrameMaker element structure in the corresponding EDD:

Element (Container): Table
General rule: Title?, Tgroup+

Element (Table): Tgroup
General rule: Thead?, Tbody, Tfoot?

Notice that the software does not create an element of type table from the CALS `table` element. The FrameMaker table model does not allow tables within tables. Because the software makes the CALS `table` element a container element, it can support multiple `tgroup` elements within the single `table` element.

In practice, many markup documents that use the CALS table model use only a single `tgroup` element within a table. In this situation, it is more natural to translate the CALS

table element as a FrameMaker element of type table and to unwrap the tgroup element. If your structure application includes a FrameMaker template, the software accommodates this behavior without read/write rules.

That is, if the definition of the Table element in your FrameMaker template is as follows:

Element (Table): Table
General rule: Title?, Thead?, Tbody, Tfoot?

then when FrameMaker opens a markup document that uses the CALS table model, it automatically unwraps the tgroup element to get this effect.

How CALS attributes translate

The CALS attributes all relate in some way to the formatting and layout of the table. Most CALS attributes become formatting properties of the resulting FrameMaker table. This reflects the fact that an EDD's format rules cannot use these properties to change the layout of the table.

Four of the CALS attributes, however, relate to the formatting of the text in a table cell. By default, the attributes align, char, charoff, and valign remain as attributes in the FrameMaker representation. This allows you to use the values of these attributes in format rules for the table and its parts.

As the last step in creating a FrameMaker document from a markup document, the software applies all the format rules in the corresponding EDD. If your markup documents use the CALS attributes that remain attributes in FrameMaker, you must add format rules to your EDD for them. If you do not add format rules for these attributes, this final step of applying format rules from the EDD removes formatting specified by those attributes in the markup document. The formatting supplied by CALS attributes that correspond only to formatting properties in FrameMaker (and not to attributes) is not overridden during this step. For more information, see "Adding format rules that use CALS attributes (CALS only)" on page 282.

How colspec and spanspec elements translate

As stated earlier, the colspec and spanspec elements do not appear in the final FrameMaker table structure. These elements exist in markup as a convenience for storing formatting information about the table part in which they occur. For example, a tgroup element may have separate colspec elements to describe characteristics of each column in the tgroup.

There are other mechanisms for storing much of this information that are more natural for FrameMaker tables. For example, instead of specifying column and row rulings for a table with colspec elements, you might choose to define particular table formats for the tables that use those elements.

If your CALS tables use colspec and spanspec elements, the software correctly interprets the attribute values of those elements even though the FrameMaker table does not retain the elements themselves. For example, the colsep attribute determines whether or not the cells of a table should have a ruling on the right side. Assume the colspec child of a tgroup element has this attribute set to 1. When processing a particular cell element

within that tgroup, the software checks to see if the cell element specifies a value for colsep. If it does not, then the software picks up the value from the colspec element of the ancestor tgroup and puts a ruling on the right. However, if the cell element has this attribute set to 0, then that cell does not have a ruling on the right.

If you want to change how FrameMaker processes any attribute of a colspec or spanspec element, you refer to the attribute as a formatting property.

On export to markup

By default, if your EDD does not use the CALS table model, FrameMaker performs no special changes when writing the table and table part elements as elements in markup. It writes only the element and attribute structure visible in the document. However, if your EDD or FrameMaker document contains elements using CALS names, the software interprets those as CALS elements unless you specify rules to the contrary. It creates the appropriate attributes corresponding to attributes and formatting information of the table.

For example, assume you have this element structure in an EDD and you save it as an SGML DTD:

Element (Container): Table
General rule: Title?, Tgroup+

Element (Table): Tgroup
General rule: Thead?, Tbody, Tfoot?

FrameMaker creates these corresponding element definitions:

```
<!ELEMENT Table - - (Title?, Tgroup+)>
<!ELEMENT Tgroup - - (Thead?, Tbody, Tfoot')>
```

If instead you have no element named Tgroup, and have this definition for Table:

Element (Table): Table
General rule: Title?, Thead?, Tbody, Tfoo?

the software creates the same element definitions as before. That is, it automatically creates the required tgroup element, even though that element was not present in the FrameMaker structure.

In addition to the element structure, the software creates the CALS attributes for the table's formatting properties and for any defined attributes.

Assume you have a document with CALS names for table elements. When you save it as markup, FrameMaker does not create spanspec elements; it puts the equivalent information in the colspec and entry elements for your table.

Modifications to the default translation

The following sections describe some of the ways you can modify the default translation of tables. If you're translating CALS tables, the default behavior is probably mostly what you

want. You'll use rules to make minor modifications such as renaming elements or changing the ruling style for a table. If you've got a different table model, you'll have to use rules more extensively to create the correspondence between markup and FrameMaker element structures.

For additional ways to modify the translation of tables, see the cross-references at the end of each section. For a summary of read/write rules relevant to translating tables, see "Tables" on page 361.

Formatting properties for tables

There are many formatting properties associated with FrameMaker tables that you might want to reference in your read/write rules. These properties are for describing general table properties such as the number of columns, straddles, or formatting properties of table cells. Some of the read/write rules that follow apply to specific formatting properties.

In translating a CALS table, the software associates some of these formatting properties with appropriate CALS table attributes by default.For example the number of columns in a CALS table becomes the `cols` attribute by default. For any other table representation, you need to write a rule to associate the appropriate formatting property with an attribute.

Properties for general table formatting

The formatting properties listed in the following table describe general table properties. More information about individual properties follows the table.

FrameMaker Property	For elements of type	CALS attribute
`column ruling`	table, cell, colspec, spanspec	`colsep`
`column width`	colspec	`colwidth`
`column widths`	table	—
`columns`	table	`cols`
`maximum height`	row	—
`minimum height`	row	—
`page wide`	table	`pgwide`
`rotate`	cell	`rotate`
`row type`	row	—
`row ruling`	table, cell, row, colspec, spanspec	`rowsep`
`table border ruling`	table	`frame`
`table format`	table	`tabstyle, tgroupstyle`

`Column ruling`: whether the specified column should having rulings on its right side. This attribute says only whether or not to have the specified ruling; it does not specify what the ruling should look like.

`Column width:` width of a single column.

`Column widths:` width of successive columns in the table. Each value is either an absolute width or a width proportional to the size of the entire table. If proportional widths are used, the `pgwide` attribute determines the table width.

For example, to specify that the first two columns are each one-quarter the size of the table and the third column is half the size of the table, you could write a rule to specify your column widths as `"25* 25* 50*"`. Valid units and abbreviations for the `column width` formatting property are:

Unit	Abbreviation
centimeter	cm
cicero	cc
didot	dd
inch	in
millimeter	mm
pica	pc (or pi)
point	pt

In FrameMaker dialog boxes, the inch unit can be represented by a double quotation mark (") as well as the `in` abbreviation. However, this convention is not supported for the `column width` formatting property. For example, `2"` is invalid as an alternative to `2in`.

`Columns:` number of columns in the table.

> **Important:** If you plan to translate markup documents to FrameMaker and your table declaration in markup does not include an attribute that corresponds to the number of columns, you *must* use the `fm property value is` rule to set a value for this property.

`Maximum height:` maximum height of a row in a table.

`Minimum height:` minimum height of a row in the table.

`Page wide:` relevant only to tables whose columns use proportional widths. In this case, the attribute indicates whether the entire table should be the width of the column in which it sits or of its text frame. If the value is unspecified or zero, the table is the width of the column; otherwise, it is the width of the enclosing text frame.

`Rotate:` how much to rotate the contents of a cell. The CALS model restricts the value of this attribute to a boolean, where 1 indicates a rotation of 90 degrees clockwise. FrameMaker extends the possible values to allow rotations of 0, 90, 180, and 270 degrees. On export, if the attribute has a positive value other than 180 or 270, the software interprets the value as 90 to be consistent with the CALS model.

Row type: whether the associated table row is a heading, footing, or body row, or the associated table cell occurs in a row of that type.

Row ruling: whether the cells of a row should having rulings on their bottom sides. This attribute says only whether or not to have the specified ruling; it does not specify what the ruling should look like.

Table border ruling: whether or not there is a ruling around the entire table. The possible values are all, top, bottom, top and bottom, sides, and none. This attribute says only whether or not to have the specified ruling; it does not specify what the ruling should look like. (You can use a rule to set the look of the ruling.)

Formatting properties for straddles

FrameMaker provides multiple ways to specify straddling information, some of which correspond to the different ways available in the CALS model.

The formatting properties listed in the following table describe straddling properties of tables. More information about individual properties follows the table.

FrameMaker Property	For elements of type	CALS Attribute
column name	cell, colspec	colname
column number	cell, colspec	colnum
end column name	cell, spanspec	nameend
horizontal straddle	cell	—
more rows	cell	morerows
span name	cell, spanspec	spanname
start column name	cell, spanspec	namest
vertical straddle	cell	—

Column name: associates a name with a given column (specified with colnum).

Column number: specifies the number of the column named with colname. Columns are numbered from left to right starting at 1. Also used to specify the column in which a cell appears.

End column name: specifies the name of a column that ends a straddle.

Horizontal straddle: how many columns this straddled cell spans.

More rows: specifies row straddling for a cell, so that the total number of rows the cell occupies is morerows+1.

Span name: names a CALS spanspec element to allow an entry element to reference it.

Start column name: specifies the name of a column that begins a straddle.

`Vertical straddle:` how many rows this straddled cell spans.

> **Important:** If you are not using the CALS table model and you want to specify straddling information on export to markup, you should use only the `horizontal straddle` and `vertical straddle` properties. The other straddle properties exist to support the alternatives available in the CALS model.

For information on how to use the formatting properties for straddles with the CALS table model, see "Attribute structure" on page 491.

Formatting properties for cell paragraph formatting

The formatting properties listed in the following table describe characteristics of a cell's paragraph format and are defined only for CALS colspecs and spanspecs. You can use format rules to map these CALS attribute values to the associated paragraph format properties.

FrameMaker Property	CALS Attribute
cell alignment character	`char`
cell alignment offset	`charoff`
cell alignment type	`align`
vertical alignment	`valign`

Note that these CALS attributes are retained in the FrameMaker document. This means you must write format rules to use these attributes to format a table, but it allows you to control the paragraph properties explicitly. For more information, see "Adding format rules that use CALS attributes (CALS only)" on page 282.

These properties only exist for `colspec` and `spanspec` elements. Other elements must use attributes to refer to this information.

`Cell alignment character:` relevant only if the `align` attribute is `char`. Determines the character on which text aligns.

`Cell alignment offset:` relevant only if the `align` attribute is `char`. Determines the location of the tab stop.

`Cell alignment type:` determines horizontal justification within a cell. Its legal values are `left`, `right`, `center`, `justify`, and `char`. If this attribute is set to `char`, the cell acts as though its autonumber were a tab character and is aligned around a character specified with the `char` attribute.

`Vertical alignment:` determines vertical positioning of the text in a cell.

Identifying and renaming table parts

If your DTD uses a table model other than the CALS model, you must identify the individual parts of the table such as the title and body rows. Additionally, you may choose to rename

these elements. If you're using the CALS table model, you don't need to identify the table parts, but you may want to use different element names in FrameMaker.

FrameMaker provides several rules for these purposes. If you include these rules, the software uses them to determine what elements to create in the EDD and to translate instances of table elements between FrameMaker and markup.

The rules for identifying and renaming table parts are as follows:

```
element "gi" is fm table element ["fmtag"];
element "gi" is fm table title element ["fmtag"];
element "gi" is fm table heading element ["fmtag"];
element "gi" is fm table body element ["fmtag"];
element "gi" is fm table footing element ["fmtag"];
element "gi" is fm table row element ["fmtag"];
element "gi" is fm table cell element ["fmtag"];
```

where *gi* is an generic identifier and *fmtag* is a FrameMaker element tag. The optional *fmtag* argument allows the element to be renamed. If *fmtag* is not specified, the name remains the same in the two representations.

If you identify a FrameMaker element as a table part, your document cannot use that element outside a table. For example, assume you have the rule:

```
element "entry" is fm table cell element "Cell";
```

The corresponding EDD contains an element `Cell`. Documents created with this EDD should not place the `Cell` element anywhere other than as a table cell. If they do so, the resultant document will be invalid.

If your DTD has an element you identify as a table element and another element you identify as a table part such as a table cell, it may not also include other elements that correspond to the intervening table parts. For information on how FrameMaker handles this, see "Omitting explicit representation of table parts" on page 285.

For information on these rules, see "element" on page 376, "is fm table element" on page 435, and "is fm table part element" on page 436.

Representing FrameMaker table properties as attributes in markup

If you use the CALS table model, FrameMaker automatically represents some table properties as attributes by default. If you use another table model, FrameMaker does not recognize any attributes as table properties.

If you have a variant of the CALS model, you can choose different names for these attributes. If you have any other table model and you want to map attributes to formatting properties, you can do so. To perform either of these tasks, use this version of the `attribute` rule:

```
attribute "attr" is fm property prop;
```

If your DTD uses a particular attribute name for the same purpose within multiple elements, you may want to use this rule as a highest-level rule to set a default. For example, assume you have four different markup elements representing different types of tables. All four elements use the attribute `numc` to represent the number of columns in the table. In this case, you would use the rule:

```
attribute "numc" is fm property columns;
```

With this rule, the software interprets the attribute `numc` as the number of columns in a table for all elements in which it occurs. If you use the same attribute for another purpose in another element, you must write a local `attribute` rule to handle it appropriately.

Alternatively, you may have only one element, `tab`, representing a table. In this case, you should restrict the association of `numc` with the `columns` property only to that element using this rule:

```
element "tab" {
   is fm table element;
   attribute "numc" is fm property columns;
}
```

With this rule, other elements can use the `numc` attribute for different purposes.

In both of these examples, the software doesn't create structure that corresponds to the `numc` attribute, but uses the attribute to read or write the appropriate information from instances of the table element.

For information on the available table formatting properties and on the CALS attributes that map to formatting properties, see "Formatting properties for tables" on page 276. For information on the rules used in these examples, see "attribute" on page 367, "element" on page 376, "is fm table element" on page 435, and "is fm property" on page 424.

Representing FrameMaker table properties implicitly in markup

A table formatting property in FrameMaker may always have the same value when applied to a particular markup element representing tables in your application. If you don't have an attribute in markup for this property, you cannot assume that the software will appropriately format the table on the basis of the element. In that case you can use the following rule to specify the value explicitly:

```
fm property prop {
   value is "propval";
}
```

where *`prop`* is the name of the property and *`propval`* is the property value.

This rule tells the software to assign a particular value to one of the formatting properties on import and not to write an attribute with the value on export.

The `fm property` rule can be used at highest level to set a default or within an `element` rule to be restricted to a single element.

For example, you may have a markup element `tab2` that represents tables with a 1-inch column followed by a 2-inch column. The `tab2` element does not use an attribute for this information but you can translate `tab2` to a FrameMaker table element as follows:

```
element "tab2" {
   is fm table element "Two Table";
   fm property columns value is "2";
   fm property column widths value is "1in 2in";
}
```

In this case, when the software encounters a start-tag for a `tab2` element on import, it creates a table element named `Two Table`. The associated table has two columns, 1 and 2 inches wide. When it encounters a `Two Table` table element on export, it writes a start-tag for a `tab2` element. It does not write attributes for the number or widths of its columns.

> **Important:** If your markup table declarations do not include an attribute that corresponds to the `columns` property and you plan to open an associated markup document in FrameMaker, you *must* use this rule to supply a value for the number of columns in the table.

For information on the rules used in this example, see "element" on page 376, "is fm table element" on page 435, and "fm property" on page 399.

Adding format rules that use CALS attributes (CALS only)

Four attributes of the CALS table model remain attributes when a markup document is read into FrameMaker. These attributes, `align`, `valign`, `char`, and `charoff`, provide information on the formatting of text within table cells. In FrameMaker, this formatting is controlled by format rules in the EDD. Keeping the attributes in the FrameMaker representation allows you to control this information explicitly.

For example, if you wanted to make use of values of the `valign` attribute, you could have this definition for a table cell:

Element (Table Cell): Entry
General rule: <TEXT>
Text format rules
1. If context is: [Valign = "Top"]
Table cell properties
Vertical alignment: Top
Else, if context is: [Valign = "Middle"]
Table cell properties
Vertical alignment: Middle
Else
Table cell properties
Vertical alignment: Bottom

For information on writing format rules, see Chapter 10, “Text Format Rules for Containers, Tables, and Footnotes.”

Working with colspecs and spanspecs (CALS only)

You may use a table model that is essentially the CALS table model but you choose to rename some of the elements, even in the DTD. If you do not use the default names for the elements that represent `colspecs` and `spanspecs`, you need to let the software know which elements to use.

The rules for identifying colspecs and spanspecs are:

```
element "gi" is fm colspec;
element "gi" is fm spanspec;
```

where *gi* is a generic identifier.

As usual with `colspec` and `spanspec` elements, the named markup element does not become an element in FrameMaker when you use these rules. Rather, its attributes are used in the creation of the table element.

For more information on these rules, see “element” on page 376, “is fm colspec” on page HIDDEN, and “is fm spanspec” on page HIDDEN.

Specifying which part of a table a row or cell occurs in

In markup your table may not have elements for rows or particular table parts such as the heading or body. Instead, the element type of a table row or cell may determine what part of the table the element goes in. For example, an element named `hrow` might only be used for a row in the heading of a table. You might not have a separate element for the heading.

If a table row does not occur inside a specified table part, the software assumes the row belongs in the table body by default. You can change this behavior using the following rule:

```
element "gi" {
   is fm table row_or_cell element;
   fm property row type value is "part";
}
```

where *gi* is a generic identifier; *row_or_cell* is one of the keywords `row` or `cell`; and *part* is one of `Heading`, `Body`, or `Footing`.

For an example of the use of these rules, see “Omitting explicit representation of table parts” on page 285.

For more information on these rules, see “element” on page 376, “is fm table part element” on page 436, and “fm property” on page 399.

Specifying which column a table cell occurs in

In markup your table may not have elements for rows or particular table parts such as the heading or body. As indicated in "Specifying which part of a table a row or cell occurs in," (the previous section), the element type of a table row or cell may determine what part of the table the element goes in. In this case, you may also need to give FrameMaker other information such as which column a table cell should be in and the fact that an element of this type indicates the start of a new row.

To tell FrameMaker which column a table cell goes in, you set the `column number` property on that element, using this rule:

```
element "gi" {
   is fm table cell element;
   fm property column number value is "n";
}
```

where *gi* is a generic identifier and *n* is an integer indicating the table column. Table columns are numbered starting with 1.

If you tell FrameMaker to put a table cell element in a particular column and there is already content in that column, FrameMaker creates a new table row to hold the element. For example, if you specify that the `term` element always occurs in column 1 and there are no vertical straddles in that column, FrameMaker creates a new table row whenever it encounters that element. For an example of this use of the column number property, see "Omitting explicit representation of table parts," next.

Your tables can have vertical straddles: the element structure of such a table reflects a vertical straddle by not having table cell elements in the straddled rows. FrameMaker cannot tell the difference between a table cell element missing because of a straddle and a table cell element missing because that cell has not yet been filled in. For this reason, it may not be enough to tell FrameMaker that an element belongs in the first column to force it to start a new row for the element. If your table can have vertical straddles and you want a particular element always to occur in a new row, you should use these rules:

```
element "gi" {
   is fm table cell element;
   fm property column number value is "n";
   reader start new row;
}
```

where *gi* is a generic identifier and *n* is an integer indicating the table column. Table columns are numbered starting with 1.

For an example of this use of these rules, see "Creating parts of a table even when those parts have no content" on page 287.

For more information on these rules, see "element" on page 376, "is fm table part element" on page 436, "fm property" on page 399, and "start new row" on page 458.

Omitting explicit representation of table parts

In markup you have the flexibility of thinking of tables as either inherently structured with rows and cells or as simply a formatting choice for some completely different element structure. In FrameMaker, though, elements must be table (and table part) elements if they are to be formatted as tables (and table parts). To facilitate formatting markup element structures as tables even when those elements do not reflect table structure, FrameMaker will create missing pieces of table structure for you.

For example, assume you have a table of terms and their definitions. Your markup can be as simple as this:

```
<glossary>
   <label>Term</label><label>Definition</label>
   <term>Mouse</term>
   <defn>A small animal</defn>
   <term>Cat</term>
   <defn>A bigger animal</defn>
   <term>Elephant</term>
   <defn>An even bigger animal</defn>
</glossary>
```

This structure does nothing to identify the rows and columns of the table. Nevertheless, you can have this structure become a table in FrameMaker by specifying mappings for the existing elements and having definitions for the missing table parts in your EDD, even though the missing table parts won't appear in your markup document.

Assume your EDD has these definitions:

Element (Table): Glossary
General rule: GlossaryHead, GlossaryBody
Text format rules
1. In all contexts.
Use paragraph format: TableCell

Element (Table Heading): GlossaryHead
General rule: GlossaryHeadRow
Text format rules
1. In all contexts.
Default font properties
Weight: Bold

Element (Table Row): GlossaryHeadRow
General rule: Label, Label

Element (Table Cell): Label
General rule: <TEXT>

Element (Table Body): GlossaryBody
General rule: GlossaryRow+

Element (Table Row): GlossaryRow
General rule: Term, Definition

Element (Table Cell): Term
General rule: <TEXT>
Text format rules
1. In all contexts.
Default font properties
Angle: Italic

Element (Table Cell): Definition
General rule: <TEXT>

And you have the following rules:

```
element "glossary" {
   is fm table element;
   fm property columns value is "2";
}

element "label" {
   is fm table cell element;
   fm property row type value is "Heading";
}

element "term" {
   is fm table cell element;
   fm property column number value is "1";
   fm property row type value is "Body";
}

element "defn" is fm table cell element "Definition";

fm element "GlossaryHead" unwrap;
fm element "GlossaryBody" unwrap;
fm element "GlossaryHeadRow" unwrap;
fm element "GlossaryRow" unwrap;
```

With these rules, the software does the following when you import the markup document containing the `glossary` example:

1. When it encounters the start-tag for `glossary`, the software creates a new 2-column `Glossary` table element. Since `glossary` does not have an attribute corresponding to the `columns` property, you had to set this value explicitly.
2. When it encounters the start-tag for the first `label` element, the software notes that this element is a table cell element and that it belongs in a table heading. However, there is not yet a table heading or row in which to put it. The software checks the definition of `Glossary` and creates the intervening `GlossaryHead` and `GlossaryHeadRow` elements. It then adds the `label` element as the first cell in the `GlossaryHeadRow`.

3. When it encounters the second `label` element, the software fills in the second column of the current heading row.
4. When it encounters the first `term` element, the software notes that this table cell element belongs in a table body. It finishes creating the heading row, checks the `Glossary` definition again, and creates the intervening `GlossaryBody` and `GlossaryRow` elements. It then adds the `term` element as the first cell in the `GlossaryRow`.
5. When it encounters the first `defn` element, the software notes that this is another table cell element and that nothing special has been said about it. The software therefore adds this element as the second cell in the first row of the table body.
6. When it encounters the second `term` element, the software notes that this table cell element is supposed to be in the first column of the table. Accordingly, it creates a new table row and puts the `term` element in its first column.
7. And so on.

In this way, the markup structure becomes the following table in a FrameMaker document:

Term	Definition
Mouse	A small animal
Cat	A bigger animal
Elephant	An even bigger animal

When this FrameMaker table is written as markup, the `fm element` rules specify that the intervening levels of table structure are not written out. Consequently, the resulting markup looks as it originally did.

For more information on the `row type` formatting property, see "Specifying which part of a table a row or cell occurs in," (the previous section).

For more information on these rules, see "element" on page 376, "fm element" on page 397, "is fm table element" on page 435, "is fm table part element" on page 436, "fm property" on page 399, and "unwrap" on page 461.

Creating parts of a table even when those parts have no content

When FrameMaker creates a table while importing a markup document, by default it creates only the table parts that have content. For example, your table definition may state that a table has a title but if the table instance in markup does not include a title, then the software does not create a title for the table.

To have the software create a special table part even if it has no content, use one of these rules:

```
reader insert table title element "fmtag";
reader insert table heading element "fmtag";
reader insert table footing element "fmtag";
```

where *fmtag* is a FrameMaker element tag.

For example, assume you have a variant of the example used in the previous section. Instead of having to specify the labels for the heading rows explicitly, the markup representation assumes that the software will put in the appropriate labels. In this case, the markup for the table is:

```
<glossary>
   <term>Mouse</term>
   <defn>A small animal</defn>
   <term>Cat</term>
   <defn>A bigger animal</defn>
   <term>Elephant</term>
   <defn>An even bigger animal</defn>
</glossary>
```

However, the intent is to have the table appear as before. In this case, your EDD definitions are similar to before. In place of this definition:

Element (Table Cell): Label
General rule: <TEXT>

you now have this definition:

Element (Table Cell): Label
General rule: <EMPTY>
Text format rules
1. **If context is:** {first}
Numbering properties
Autonumber format: Term
Else
Numbering properties
Autonumber format: Definition

This definition says that if a `Label` element occurs as the first child of its parent (the first column in a row), then the word “Term” appears at the beginning of its (otherwise empty) text. The word “Definition” appears in all other columns.

With the modified definitions, you use modified rules. Since `label` is no longer part of the markup element structure, you replace these rules:

```
element "glossary" {
   is fm table element;
   fm property columns value is "2";
}

element "label" {
   is fm table cell element;
   fm property row type value is "Heading";
}

fm element "GlossaryHead" unwrap;
fm element "GlossaryHeadRow" unwrap;
```

with these rules:

```
element "glossary" {
   is fm table element;
   fm property columns value is "2";
   reader insert table heading element "GlossaryHead";
}

fm element "GlossaryHead" drop;
```

For more information on these rules, see “element” on page 376, “fm element” on page 397, “is fm table element” on page 435, “is fm table part element” on page 436, “fm property” on page 399, “reader” on page 453, “insert table part element” on page 410, “drop” on page 373, and “unwrap” on page 461.

Specifying the ruling style for a table

The `frame`, `colsep`, and `rowsep` CALS attributes determine whether or not a ruling should be applied to the appropriate part of a table. These attributes are all booleans; that is, they specify simply whether or not a ruling should be applied. FrameMaker supports several ruling styles. To specify the ruling style for the entire table, you can use the following rule:

```
reader table ruling style is "style";
```

where *`style`* is the name of a ruling style. This rule sets the ruling style for all tables. For example, to set the outer borders of all tables that have outer borders to use a thick ruling style, you would use this rule:

```
reader table ruling style is "Thick";
```

A ruling set in this manner is considered as custom ruling and shading by the software.

For information on formatting tables, see the FrameMaker user’s manual. For more information on these rules, see “reader” on page 453 and “table ruling style is” on page 461 of this manual.

Exporting table widths proportionally

When you export a table, the software writes the width of the columns as absolute measurements by default. If you want to use proportional widths instead, you can use these rules:

```
writer use proportional widths;
writer proportional width resolution is "value";
```

where *value* is an integer. The proportions of all columns in a table add to *value*. If you do not specify the `proportional width resolution` rule, the default is 100; that is, the proportional widths of all columns add to 100. If you do not specify the `use proportional widths` rule, the software writes absolute widths for all tables.

For example, assume you use the CALS table model and you've added these rules:

```
writer use proportional widths;
writer proportional width resolution is "4";
```

If you export a FrameMaker document containing a 3-column table whose columns are, respectively, 1 in, 1 in, and 2 in wide, the software writes the following `colspec` start-tags for the table:

```
<colspec colname = "1" colnum = "1" colsep = "0" colwidth = "1*">
<colspec colname = "2" colnum = "2" colsep = "0" colwidth = "1*">
<colspec colname = "3" colnum = "3" colsep = "0" colwidth = "2*">
```

If the table's actual column widths do not add to the resolution, FrameMaker rounds the values as necessary. For example, if the above table columns were actually 0.75 in, 1.2 in, and 2.2 in, the software would still write the same `colspec` start-tag.

You can use the `use proportional widths` rule with any attribute that has been associated with the `column widths` property, not just the CALS attributes.

For more information on these rules, see "writer" on page 467, "use proportional widths" on page 463, and "proportional width resolution is" on page 452.

Creating vertical straddles

FrameMaker provides two rules for you to use if your table structure defines rows that are always straddled. In an `element` rule for a table cell, you can use this rule:

```
reader start vertical straddle "name";
```

In an `element` rule for a table row, you can use this rule:

```
reader end vertical straddle "name1 . . . nameN";
reader end vertical straddle "name1 . . . nameN" before this row;
```

where *name* identifies the element that starts a vertical straddle, and each $name_i$ is a previously named straddle that ends with this element.

For example, a book on marine life might have tables of fish, including the general category and several subtypes of that category, with locations in which you can find the subtypes. Here's an example of such a table:

General type	Subtype	Location
Lionfish	Clearfin	Egypt
	Ocellated	French Polynesia
	Spotfin	Papua New Guinea
Eel	Blue ribbon	Fiji
	Moray	Pretty much everywhere

In your markup representation, you may choose to use an element structure such as the following:

```
<range>
   <type>Lionfish</type>
      <subtype>Clearfin</subtype><loc>Egypt</loc>
      <subtype>Ocellated</subtype><loc>French Polynesia</loc>
      <subtype>Spotfin</subtype><loc>Papua New Guinea</loc>
   <type>Eel</type>
      <subtype>Blue ribbon</subtype><loc>Fiji</loc>
      <subtype>Moray</subtype><loc>Pretty much everywhere</loc>
</range>
```

The markup representation assumes that the formatting software knows that this should become a 3-column table and that a cell containing a `type` element straddles all the rows that contain the following `subtype` elements. The straddle ends just before the next `type` element.

To produce the table above, include these definitions in your EDD:

Element (Table): Range
  General rule: RangeHead, RangeBody
  Text format rules
    1. In all contexts.
      Use paragraph format: TableCell

Element (Table Heading): RangeHead
  General rule: RangeHeadRow
  Text format rules
    1. In all contexts.
      Default font properties
        Weight: Bold

Element (Table Row): RangeHeadRow
  General rule: Label, Label, Label

Element (Table Cell): Label
General rule: <EMPTY>
Text format rules
1. If context is: {first}
Numbering properties
Autonumber format: General type
Else, if context is: {last}
Numbering properties
Autonumber format: Location
Else
Numbering properties
Autonumber format: Subtype

Element (Table Body): RangeBody
General rule: RangeRow+

Element (Table Row): RangeRow
General rule: Type?, Subtype, Location

Element (Table Cell): Type
General rule: <TEXT>

Element (Table Cell): Subtype
General rule: <TEXT>

Element (Table Cell): Location
General rule: <TEXT>

And the following rules in your read/write rules document:

```
element "range" {
   is fm table element;
   fm property columns value is "3";
   reader insert table heading element "RangeHead";
}

element "type" {
   is fm table cell element;
   fm property column number value is "1";
   fm property row type value is "Body";
   reader start vertical straddle "Type";
   reader end vertical straddle "Type" before this row;
   reader start new row;
}

element "subtype" {
   is fm table cell element;
   fm property column number value is "2";
}
```

```
element "loc" {
   is fm table cell element "Location";
   fm property column number value is "3";
}

fm element "RangeHead" drop;
fm element "RangeBody" unwrap;
fm element "RangeRow" unwrap;
```

This example builds on information in “Specifying which part of a table a row or cell occurs in” on page 283, “Omitting explicit representation of table parts” on page 285, and “Creating parts of a table even when those parts have no content” on page 287.

For more information on these rules, see

- “element” on page 376
- “fm element” on page 397
- “is fm table element” on page 435
- “is fm table part element” on page 436
- “fm property” on page 399
- “reader” on page 453
- “insert table part element” on page 410
- “end vertical straddle” on page 379
- “start vertical straddle” on page 460
- “start new row” on page 458
- “drop” on page 373
- “unwrap” on page 461

Using a table to format an element as a boxed set of paragraphs

The formatting associated with your documents may require that the paragraphs in an element appear in a completely or partially boxed area. In SGML, assume you have the following element declaration:

```
<!element note - - ((#PCDATA | emphasis | code)+)>
```

To format this element as a boxed paragraph in FrameMaker, you use a one-cell table with appropriately defined ruling properties. The corresponding EDD looks as follows:

Element (Table): Note
General rule: NoteBody
Initial table format
1. In all contexts.
Table Format: NoteTable
Text format rules
1. In all contexts.
Use paragraph format: note

Element (Table Body): NoteBody
General rule: NoteRow

Element (Table Row): NoteRow
General rule: NoteCell

Element (Table Cell): NoteCell
General rule: (<TEXT> | Emphasis | Code)+

The `NoteTable` table format can specify ruling that boxes the paragraph. For example, the EDD for this manual uses this technique to format important information such as the following:

> This boxed paragraph is implemented as a one-cell table.

Using read/write rules, to you can translate a single markup element that needs to be formatted as one or more boxed paragraphs into a one-cell table in FrameMaker. To create the element definitions above, use the following rules:

```
element "note" {
   is fm table element;
   fm property columns value is "1";
}
fm element "NoteBody" unwrap;
fm element "NoteRow" unwrap;
fm element "NoteCell" unwrap;
```

With these rules, FrameMaker creates the `Note` table element when it encounters a `note` element while importing a markup document. The next thing it encounters is text to go into the table, but text can't directly be put into a table—it must go in a table cell—so the intervening parts of the table need to be supplied. Since the markup doesn't specify any child elements, the software uses the structure specified in the EDD for a `Note` element. That is, it creates `NoteBody`, `NoteRow`, and `NoteCell` elements and places the text in the `NoteCell` element.

On export, the software unwraps the `NoteBody`, `NoteRow`, and `NoteCell` elements and writes out only the single `note` element.

For information on these rules, see “element” on page 376, “fm element” on page 397, “is fm table element” on page 435, “fm property” on page 399, and “unwrap” on page 461. For information on creating table formats, see the FrameMaker user’s manual.

Creating tables inside other tables

The FrameMaker table model does not allow you to place a table directly inside another table. To put a table inside another table, you must put the inner table inside an anchored frame. Note that the table inside the anchored frame is in a different text flow than the outer table. For this reason, if your EDD allows for this situation and you need to export such tables to markup, you’ll need to write a structure API client to do so.

For information on writing structure API clients, see the*Structure Import/Export API Programmer’s Guide*.

Rotating tables on the page

In FrameMaker, you cannot directly specify that an entire table be rotated on the page. If you need to rotate tables, the *FrameMaker User Guide* manual suggests two approaches: you can have the table format start at the top of a page and apply a rotated master page. Or you can put the table inside an anchored frame and rotate the table inside the frame.

The first method does not require special processing on export to markup. Since the second method places the table in a separate text flow, you must write a structure API client to perform export to markup if you use it.

For information on writing structure API clients, see the*Structure Import/Export API Programmer’s Guide*.

19 Translating Graphics and Equations

FrameMaker provides a set of tools for creating graphics or equations. It also provides tools for importing graphic objects created with another software package into a FrameMaker document. Markup, on the other hand, does not standardize the representation of either graphics or equations; each DTD can treat them differently.

In FrameMaker equations are very similar to graphics. Both are placed within anchored frames, and the anchored frames contain one or more objects—for graphics these are graphic objects, and for equations they are terms of the equation. In the context of structure and import/export of markup, an equation is treated as a single graphic object within an anchored frame. FrameMaker does nothing to represent the equation's terms as any type of structure. You cannot use rules to describe the internal structure of equations. However, you can write a structure API client to support a completely different model for equations (such as MathML).

FrameMaker has a default set of element and attribute definition list declarations for representing graphics and equations as elements. You can use read/write rules to support variations of the default representation, whether you start with an EDD or a DTD.

In this chapter

This chapter describes how FrameMaker translates graphics and equations and how you can change that translation. In the outline below, click a topic to go to its page.

How FrameMaker translates graphics and equations by default:

- "On export to markup" on page 300
- "On import to FrameMaker" on page 307

Some ways you can change the default translation:

- "Identifying and renaming graphic and equation elements" on page 309
- "Exporting graphic and equation elements" on page 309
- "Representing the internal structure of equations" on page 311
- "Renaming markup attributes that correspond to graphic properties" on page 311
- "Omitting representation of graphic properties in markup" on page 312
- "Omitting optional elements and attributes from the default DTD declarations" on page 313
- "Specifying the data content notation on export" on page 313

- "Changing the name of the graphic file on export" on page 314
- "Changing the file format of the graphic file on export" on page 316
- "Creating graphic files on export" on page 306
- "Specifying the entity name on export" on page 318
- "Changing how FrameMaker writes out the size of a graphic" on page 319

Default translation

FrameMaker has well-defined representations for graphics and equations. They are given element structure as *graphic elements* and *equation elements*. You can create a graphic in an external tool and then import it into FrameMaker, you can use the software's tools to create the graphic, or you can combine methods. Whatever the method, the software puts the graphic into an anchored frame in the document. For equations, you create an equation using the software's equations tools. FrameMaker treats the equation as a single graphic object inside an anchored frame and exports this anchored frame to markup.

Supported graphic file formats

FrameMaker supports multiple graphic file formats for graphic objects created in an external tool, such as the QuickDraw PICT (PICT) format or the Computer Graphics Metafile (CGM) format. If you have a document with a graphic in a file format that the software doesn't support, you can supply your own graphic filter to allow the graphic to be processed. You can have such a document regardless of whether you are starting from markup or from FrameMaker.

The available export graphic formats for FrameMaker 7.0 are:

Code	Meaning
CDR	CorelDRAW
CGM	Computer Graphics Metafile
DIB	Device-independent bitmap (Windows import only)
DRW	Micrografx CAD
DWG	Autocad Drawing
DXF	Autodesk Drawing eXchange file (CAD files)
EMF	Enhanced Metafile (Windows)
EPSB	Encapsulated PostScript Binary (Windows)
EPSD	Encapsulated PostScript with Desktop Control Separations (DCS)
EPSF	Encapsulated PostScript (Macintosh)
EPSI	Encapsulated PostScript Interchange
FRMI	FrameImage

Code	Meaning
FRMV	FrameVector
G4IM/GP4	CCITT Group 4 to Image
GEM	GEM file (Windows)
GIF	Graphics Interchange Format (Compuserve)
HPGL	Hewlett-Packard Graphics Language
IGES	Initial Graphics Exchange Specification (CAD files)
IMG4	Image to CCITT Group 4 (UNIX)
JPG, JPE (JFIF)	Joint Photographer Experts Group (actual file format is JPEG File Interchange Format
MooV	QuickTime Movie
OLE	Object Linking and Embedding Client (Microsoft)
PCX	PC Paintbrush
PICT	QuickDraw PICT
PNG	Portable Network Graphic
PNTG	MacPaint
SNRF	Sun Raster File
SRGB	SGI RGB
SVG	Scalable Vector Graphics
TIFF	Tag Image File Format
WMF	Windows Metafile
XBM	X BitMap (Unix only)
XWD	X Windows System Window Dump file

General import and export of graphic elements

Markup does not standardize the representation of graphics or equations in a DTD. Consequently, their representation can be unique to a DTD. Without rules, the software can't identify elements and attributes representing a graphic or an equation when creating an EDD from a DTD. However, you can easily use the element and attribute structure provided with FrameMaker, or a variant of that structure.

Even though markup does not have a standard representation for graphics, there are some commonly used frameworks and FrameMaker provides support for two of them—the read/write rules work well to modify them. If your element and attribute structure matches one of these frameworks, it should be relatively straightforward for you to import or export graphics or equations. However, if your DTD uses a radically different framework, you'll have to write a structure API client.

Basically, FrameMaker assumes that in markup a graphic or equation is an empty element with either an `ENTITY` attribute identifying an external data entity that is the actual graphic

or a CDATA attribute whose value is a filename containing the graphic. Other attributes can represent properties of the FrameMaker anchored frame in which the graphic sits inside a FrameMaker document.

> **XML:** The XML specification states that an external graphic is a non-parsed entity and must be referenced via an ENTITY or ENTITIES attribute in an empty element. This corresponds very well with the preferred framework for representing graphics in a FrameMaker structure application.

If the graphic file is specified in an entity declaration, the entity name is always kept with the graphic inset in the FrameMaker document. Note, however, that there is no way to see the entity name by inspecting a graphic inset. On export, this entity name becomes the value of the entity attribute of the graphic element. If the graphic's entity declaration is in the internal DTD subset of the imported markup document, FrameMaker stores the information so it can recreate the entity declaration on export. For information on how FrameMaker saves the entity definition, see "Importing graphic entities" on page 307. For information on how FrameMaker exports entity declarations, see "Exporting entity declarations" on page 305. For information on how FrameMaker exports graphic files, see "Creating graphic files on export" on page 306.

An external data entity has an associated notation, designed to tell the structure application how to render the entity data. When the software reads a markup document, rather than storing this information in attributes or variables in FrameMaker, it stores the information directly in the graphic's associated anchored frame. This results in fewer attributes on the FrameMaker element than were on the markup element. When exporting a document to markup, the software recreates this information in the attributes and entities it writes.

You can use rules to modify some of what FrameMaker writes on export, such as the name of the entity or which, if any, of the graphic's facets get written as files. There are few rules that relate specifically to modifying what the software does to graphics on import.

On export to markup

Some properties of FrameMaker graphics and equations are not explicit in their element and attribute structure and the software translates these properties as attributes in markup. These properties are named so that you can refer to them in rules. For example, the `align` attribute corresponds to the `alignment` property and indicates how an anchored frame is aligned on the page.

Also, in some circumstances FrameMaker will write out a new graphic file when exporting a document to markup. For more information, see "Creating graphic files on export" on page 306

Text of default graphics and equations declarations

> **XML and SGML:** The following examples show declarations for XML. For SGML, the instances of NMTOKEN can be expressed as NUMBER or NAME

The default DTD declarations for graphics and equations are described in detail in the following sections. In summary, the following examples show the default declarations.

```
<!-- graphic and equation elements-->
<!ELEMENT mygraphic   EMPTY>
<!ELEMENT myequation   EMPTY>

<!--Attributes for equations-->
<!ATTLIST myequation
   entity      ENTITY  #IMPLIED <!-- graphic entity -->
   file        CDATA   #IMPLIED <!-- graphic file, SGML only -->
   align       NMTOKEN #IMPLIED <!--frame alignment on page-->
   angle       CDATA   #IMPLIED <!--frame angle in degrees-->
   bloffset    CDATA   #IMPLIED <!--frame baseline offset-->
   cropped     NUMBER  #IMPLIED <!--nonzero for cropped-->
   float       NUMBER  #IMPLIED <!--nonzero for floating-->
   height      CDATA   #IMPLIED <!--frame height-->
   nsoffset    CDATA   #IMPLIED <!--frame near-side offset-->
   position    NAME    #IMPLIED <!--frame position-->
   width       CDATA   #IMPLIED <!--frame width-->
attribute_declarations_specific_to_this_equation_element
>

<!--Attributes for graphics-->
<!ATTLIST mygraphic
   entity   ENTITY  #IMPLIED
   file     CDATA   #IMPLIED <!-- graphic file, SGML only -->
   align    NMTOKEN #IMPLIED <!--frame alignment on page-->
   angle    CDATA   #IMPLIED <!--frame angle in degrees-->
   bloffset CDATA   #IMPLIED <!--frame baseline offset-->
   cropped  NMTOKEN #IMPLIED <!--nonzero for cropped-->
   float    NMTOKEN #IMPLIED <!--nonzero for floating-->
   height   CDATA   #IMPLIED <!--anchored frame height-->
   nsoffset CDATA   #IMPLIED <!--frame near-side offset-->
   position NMTOKEN #IMPLIED <!--frame position-->
   width    CDATA   #IMPLIED <!--frame width-->
   dpi      NMTOKEN #IMPLIED <!--ignored if impsize specified-->
   impang   CDATA   #IMPLIED <!--angle in frame, in deg-->
   impby    (ref|copy) #IMPLIED <!--import by reference or copy-->
   impsize  CDATA   #IMPLIED <!--import size (width & height)-->
   sideways NMTOKEN #IMPLIED <!--1 if object turned sideways-->
   xoffset  CDATA   #IMPLIED <!--horizontal offset in frame-->
   yoffset  CDATA   #IMPLIED <!--vertical offset in frame-->
attribute_declarations_specific_to_this_graphic_element
>
```

Element and attribute structure This set of declarations represents graphics and equations using elements with a declared content of `EMPTY` and two primary attributes, `entity` and `file`. For a given element, the software writes only one of `entity` or `file`. The `entity` attribute is of type `ENTITY` and identifies an external data entity containing the graphic or equation. The `file` attribute is of type `CDATA` and its value is the name of a file containing the graphic or equation.

> **XML:** For XML you should use the `entity` attribute to treat a graphic file as an unparsed entity.

When FrameMaker encounters a graphic or equation on export, it writes a start-tag for an empty element, including values for its attributes, as appropriate. Under some circumstances, it also writes a file containing the graphic or equation itself. You can use read/write rules or a structure API client to change these behaviors. For information about exporting graphic or equation files, see "Creating graphic files on export" on page 306. For information on facets, see the FrameMaker user's manual.

Graphic and equation elements have the same set of attributes describing common properties of anchored frames. Graphic elements have additional attributes for properties relevant only to graphics created outside FrameMaker. Finally, your EDD might define other attributes for a particular graphic or equation element. The software exports these attributes as well.

Entity and file attributes The default declarations include both an `entity` and a `file` attribute for each markup element that corresponds to a graphic or equation. When exporting a FrameMaker document, the software stores the location of the graphic file in one or the other of those attributes, depending on which is present in the markup element declaration.

If the markup element has an `entity` attribute defined, then on export FrameMaker writes a value for the `entity` attribute in markup. If there is no corresponding entity in the structure application's DTD, it also generates the corresponding entity declaration in the document's internal DTD subset. By default, FrameMaker doesn't generate a public identifier; you must supply a structure API client to do so.

If a markup element has a `file` attribute but no `entity` attribute, the software writes a value for the `file` attribute. On import, if the markup element has no value for the `entity` attribute, the value of the `file` attribute is used. Otherwise, the `entity` value is used.

Anchored frame properties All markup elements corresponding to graphics and equations have the following implied attributes that supply information about the anchored frame containing the graphic or equation:

- `align` corresponds to the `alignment` property and indicates the anchored frame's horizontal alignment on the page. Its possible values and the corresponding FrameMaker property values are as follows:

Attribute value	Property value
`aleft`	`align left`
`acenter`	`align center`
`aright`	`align right`
`ainside`	`align inside`
`aoutside`	`align outside`

- `angle` corresponds to the `angle` property and indicates an angle of rotation for the anchored frame containing the graphic. The value is assumed to represent a number. The software interprets this attribute as a number of degrees of rotation. You must specify exact multiples of 90 degrees. Otherwise, the value is ignored and the graphic is imported at 0 degrees (default). For example, if 89 degrees is specified, the graphic imports at 0 degrees.
- `bloffset` corresponds to the `baseline offset` property and indicates how far from the baseline of a paragraph to place an anchored frame. The value is assumed to represent a number. If not supplied, the value is `0`. The `bloffset` attribute is relevant only for anchored frames whose `position` attribute is one of `inline`, `sleft`, `sright`, `snear`, or `sfar`.
- `cropped` corresponds to the `cropped` property and indicates whether a wide graphic should be allowed to extend past the margins of the text frame. The value is either `0` or `1`. If not supplied, the value is `1`, indicating that the graphic should not extend past the margins. The `cropped` attribute is relevant only for anchored frames whose `position` attribute is one of `top`, `below`, or `bottom`.
- `float` corresponds to the `floating` property and indicates whether the graphic should be allowed to float from the paragraph to which it is attached. The value is `0` or `1`. If not supplied, the value is `0`, indicating that the graphic must stay with the paragraph. The `float` attribute is relevant only for anchored frames whose `position` attribute is one of `top`, `below`, or `bottom`.
- `height` corresponds to the `height` property and indicates the height of the anchored frame. If not supplied, the value for a single imported graphic object is the sum of the height of the object plus twice the value of the `yoffset` attribute. For all other graphics and for equations, the value is the height of the object.
- `nsoffset` corresponds to the `near-side offset` property and indicates how far to set a frame from the text frame to which the frame is anchored. The value is assumed to represent a number. If not supplied, the value is `0`. The `nsoffset` attribute is relevant only for anchored frames whose `position` attribute is one of `sleft`, `sright`, `snear`, or `sfar`.

- `position` corresponds to the `position` property and indicates where on the page to put the anchored frame. If not supplied, the value is `below`. Possible values of `position` and the corresponding FrameMaker property values are as follows:

Attribute value	Property value
`inline`	`inline`
`top`	`top`
`below`	`below`
`bottom`	`bottom`
`sleft`	`subcol left`
`sright`	`subcol right`
`snear`	`subcol nearest`
`sfar`	`subcol farthest`
`sinside`	`subcol inside`
`soutside`	`subcol outside`
`tleft`	`textframe left`
`tright`	`textframe right`
`tnear`	`textframe nearest`
`tfar`	`textframe farthest`
`tinside`	`textframe inside`
`toutside`	`textframe outside`
`runin`	`run into paragraph`

- `width` corresponds to the `width` property and indicates the width of the anchored frame. If not supplied, the value for a single imported graphic object is the sum of the width of the object plus twice the value of the `xoffset` attribute. For all other graphics and for equations, the value is the width of the object.

For more on these properties, see the *FrameMaker User Guide* for information about anchored frames.

Other graphic properties Markup elements corresponding to graphic elements can have the following additional implied attributes:

- `alt-text` corresponds to the `Alternate` property and provides a string to display when the XML formatting software cannot display the graphic.
- `dpi` corresponds to the `dpi` property and indicates how to scale an imported graphic object. The software ignores this attribute if it finds a value for the `impsize` attribute. It produces only one of the attributes—`dpi` or `impsize`. The value of the `dpi` attribute is a number. If not supplied, the value is `72`.
- `impang` corresponds to the `import angle` property and indicates an angle of rotation for the graphic inside its anchored frame. The value is assumed to represent a number.

FrameMaker interprets this attribute as a number of degrees of rotation. If not supplied, the value is `0.0`.

- `impsize` corresponds to the `import size` property and indicates the size of the imported graphic object by specifying a width and height. This property is set by choosing the option Fit in Selected Rectangle from the Imported Graphic Scaling dialog box. If not supplied, a `dpi` attribute value must be supplied.
- `impby` corresponds to the `import by reference or copy` property and indicates whether an imported graphic object remains in a separate file or is incorporated in the FrameMaker document on import from markup. The value is either `ref` or `copy`. If not supplied, the value is `ref`, indicating that the object should not be copied into the document.
- `rasterdpi` corresponds to the `rasterdpi` property of SVG graphics and indicates how to scale the FrameImage facet of an imported SVG graphic. It also determines the dpi when converting SVG to a raster format on export. The value of the `dpi` attribute is a number. If not supplied, the value is `72`.
- `sideways` corresponds to the `sideways` property and indicates whether the imported graphic should be inverted around its vertical axis. The value is `0` or `1`. If not supplied, the value is `0`, indicating that the graphic shouldn't be inverted.
- `xoffset` corresponds to the `horizontal offset` property and indicates how far the graphic object is offset from the right and left edges of the anchored frame. The value is assumed to represent a number. If not supplied, the value is `6.0pt`.
- `yoffset` corresponds to the `vertical offset` property and indicates how far the graphic object is offset from the top and bottom edges of the anchored frame. The value is assumed to represent a number. If not supplied, the value is `6.0pt`.

For more on these properties, see the *FrameMaker User Guide* for information about importing graphics.

Exporting entity declarations

When importing a markup document, use read/write rules to import markup elements as graphic elements or equations. If the graphic file is specified by an entity declaration, the software stores enough information about the entity declaration to recreate it on export.

If the entity declaration was made in the internal DTD subset of the document instance, that information is stored on the Entity Declarations reference page of the FrameMaker document. On export, the software will use this information to recreate the entity declarations in the resulting markup document. For more about saving entity declaration information in a FrameMaker document, see "Importing graphic entities" on page 307.

On export to markup, the software ensures the appropriate entities are in the document instance as follows. Unless otherwise specified, this table assumes the graphic element's declaration in markup includes an `entity` attribute:

If:	FrameMaker:
The graphic filename and entity name match an entity declaration in the structure application's DTD...	Uses the matching entity name as the value of the graphic element's `entity` attribute in markup.
The graphic filename matches an entity declaration on the Entity Declarations reference page...	Declares the entity in the DTD subset of the resulting markup document. It uses the entity name that was stored with the graphic inset for the entity declaration and for the `entity` attribute in the resulting graphic element in markup.
The graphic filename matches no entity declarations, or the graphic inset has no entity name associated with it...	Generates an entity declaration in the DTD subset of the resulting document instance. It names the entity graphic1, graphic2, etc. It writes the entity name to the `entity` attribute in the resulting graphic element.
The SGML graphic element does not include an `entity` attribute...	Does not generate an entity declaration. The filename for the graphic file is used as the value for the `file` attribute of the SGML graphic element.

Creating graphic files on export

For graphics imported by reference, the software uses the same file for the markup document as it does for the FrameMaker document. On export, it creates new files for graphic and equation elements that meet any of the following conditions:

- The graphic file was imported by copy.
- The user changed the graphic content in any way while editing the document in FrameMaker. This includes adding graphic content via the graphics palette, or importing an additional file into the anchored frame. Note that the user may delete the existing graphic file and import another one. If the new file matches a file in the DTD's entity declarations, or it matches a file on the Entity Declarations reference page, the exported markup will refer to this newly corresponding entity.

For each one of such graphics and equations, the software creates a new graphic file. For information on whether the software references the file via an entity or via the file attribute of the graphic element in markup, see "Exporting entity declarations," (the previous section).

A graphic element may be an anchored frame containing only a single imported graphic object. If so, it is likely to have a single facet that isn't one of the software's internal facets. In this case, the written file is in the graphic format indicated by the facet. For all other graphics and for equations, the written file is in CGM format.

If FrameMaker exports the single facet of a graphic element, the software exports the file in the indicated format and uses the facet name as the notation name. In all other cases, the software exports the file in CGM format and its notation name is `CGM`.

For example, assume you have an instance of the `Graph` graphic element that contains graphics you created with FrameMaker's graphics tools. By default, the software creates the following markup entity for it:

```
<!ENTITY graph1 SYSTEM "graph1.cgm" NDATA cgm>
```

Also, the `entity` attribute of the graphic element has a value of `graph1` to correspond with the above entity.

On import to FrameMaker

In the absence of read/write rules, FrameMaker cannot identify the elements and attributes of a markup document that correspond to a graphic or equation. Therefore, it translates them as container elements in the EDD. You must supply rules and perhaps a structure API client to reflect the appropriate structure.

If your DTD uses the default graphic and equation declarations described in the preceding sections, the only rule you need is one to identify the element as a graphic or equation. If you do so, the attributes translate automatically. The translation occurs either if you started by creating your DTD from an EDD or if you started with an existing DTD without declarations for graphics or equations and added the default declarations. For a description of these declarations, see "Text of default graphics and equations declarations" on page 300.

Your markup document can use either the `entity` attribute or (for SGML) the `file` attribute to specify a graphic or equation. If a single element has specified values for both attributes, the software uses the value of the `entity` attribute, ignoring the value of the `file` attribute. FrameMaker accepts the name of any external data entity as a value for `entity`, regardless of whether the entity is declared to be `CDATA`, `SDATA`, or `NDATA`.

Importing graphic entities

If the graphic element in markup uses the `entity` attribute to identify the graphic file, then the actual graphic file must be identified in an entity declaration. FrameMaker will read the markup, importing the graphic file by reference into an anchored frame. If the entity declaration is not in the main DTD, then it should be in the document's internal DTD subset. In that case, FrameMaker saves information about the entity declaration on the *Entity Declarations* reference page of the resulting document.

The software saves the entity name with the imported graphic inset. The software uses the entity name, plus the information on the *Entity Declarations* reference page, to recreate entity declarations when exporting to markup. For graphic entities the software can use the same graphic file for import and export under most circumstances. For information about exporting entity declarations, see "Exporting entity declarations" on page 305. For

information about when the software does and does not use the same graphic file, see "Creating graphic files on export" on page 306.

Graphic attributes and properties

The attributes defined in the default declarations for graphics and equations specify properties for the graphic or equation object in FrameMaker. On import, none of these attributes translate to FrameMaker element attributes. They are all saved as properties of the object or the graphic element's anchored frame.

If the graphic is a MIF file, and the MIF file contains an anchored frame, the MIF will also have anchored frame property values. The attributes specified in the markup document take precedence over the anchored frame property values of the MIF file. However, the object properties specified in the MIF file take precedence over the values specified in the markup document. If a MIF file doesn't contain an anchored frame and the graphic element's start-tag doesn't provide attribute values, then the software uses default values in creating the anchored frame.

Graphic file formats

FrameMaker expects the graphic file to be in one of the formats it supports. For the list of graphic file formats FrameMaker supports, see "Supported graphic file formats" on page 298.

If the file is in a format the software does not support, you may be able to add a filter to support that format. For information on the graphic filters included with the software and on how to add a new one, see the online manual about using filters for Frame products.

If the graphic is in a MIF file, FrameMaker assumes that the first anchored frame or equation on a body page in the file is the content of the element. Some filters create MIF files in which the graphic objects aren't put into an anchored frame. When it reads such a file, the software processes all the graphic objects on the first body page as a single anchored frame.

Modifications to the default translation

The read/write rules for graphics and equations allow you, among other things, to:

- identify which elements in markup correspond to graphic or equation elements
- associate attributes markup with FrameMaker formatting properties
- or specify the graphic file format for an exported graphic

The rules for equations and for graphics of various sorts are very similar. Typically, you start with an element, identify it either as a graphic or an equation, indicate what to do with attributes that represent formatting information, and indicate how to treat the element under different export circumstances.

The following sections describe some of the ways you can modify the default translation of graphics and equations. The first two procedures are relevant for translating any graphic or

equation elements. For additional ways to modify the translation of graphics and equations, see the cross-references at the end of each section.

For a summary of read/write rules relevant to translating graphics, see "Graphics" on page 358. For a summary of read/write rules relevant to translating equations, see "Equations" on page 357. For a list of the formatting properties associated with translating graphics and equations, see "Text of default graphics and equations declarations" on page 300.

Identifying and renaming graphic and equation elements

If you create or update your EDD from an existing DTD, or if you want to rename elements on import or export, you must use a rule to identify which markup elements correspond to graphic or equation elements in FrameMaker. To do so, use one of these rules:

```
element "gi" is fm graphic element ["fmtag"];
element "gi" is fm equation element ["fmtag"];
```

where *gi* is a generic identifier and the optional *fmtag* argument allows renaming the element. If *fmtag* is not specified, the name remains the same in the two representations. For example, to specify that the markup element `pict` corresponds to the graphic element `Picture` in FrameMaker, use this rule:

```
element "pict" is fm graphic element "Picture";
```

For information on the rules used in this example, see "element" on page 376, "is fm equation element" on page 420, and "is fm graphic element" on page 422.

Exporting graphic and equation elements

FrameMaker supplies several rules for modifying the software's behavior when exporting a graphic or equation element. These rules are explained in later sections of this chapter. They occur as subrules of a rule that also indicates the type of graphic or equation being exported. There are three rule constructions for this purpose:

```
element "gi" {
   is fm equation element ["fmtag"];
   writer equation { subrules }
}
element "gi" {
   is fm graphic element ["fmtag"];
   writer facet "facet" { subrules }
}
element "gi" {
   is fm graphic element ["fmtag"];
   writer anchored frame { subrules }
}
```

where *gi* is a generic identifier, *fmtag* is an optional FrameMaker element tag, *subrules* are described later, and *facet* is a graphic facet.

Use the first of these constructions to specify how the software exports an equation element under all circumstances.

Use the second construction to specify how it exports a graphic element when that graphic element contains only a single facet with the name specified by *facet*. This corresponds to the situation where the graphic element is an anchored frame containing only a single imported graphic object whose original file was in the *facet* graphic format.

Use the third construction to tell it how to export a graphic element under all other circumstances. You can use the `facet` construction multiple times if you want the software to treat file formats differently.

For example, assume you use the `Graphic` element for all graphic elements. If the graphic contains any single facet, you assume the graphic was imported as an entity and you want the default behavior. However, if the author used FrameMaker graphic tools to create the objects in the graphic element, you want the file written in QuickDraw PICT format.

To accomplish all this, you would use this rule:

```
element "graphic" {
   is fm graphic element;
   writer anchored frame export to file "$(entity).pic" as
      "PICT";
}
```

Because the entities specify graphic files that are unchanged, the software doesn't create new graphic files for those elements. However, if the author created a graphic using the FrameMaker graphic tools, there is no corresponding entity. The software will write the graphic file in PICT format, and create a corresponding graphic entity. The markup graphic element will refer to that entity via the `entity` attribute.

For more information on these export options, see "Creating graphic files on export" on page 306 and "Changing the file format of the graphic file on export" on page 316. For information on these rules, see

- "anchored frame" on page 365
- "facet" on page 394
- "equation" on page 385
- "element" on page 376
- "is fm equation element" on page 420
- "is fm graphic element" on page 422
- "export to file" on page 389
- "writer" on page 467

Representing the internal structure of equations

As mentioned earlier in this chapter, FrameMaker treats an equation in the same way it does a graphic element. That is, on export to markup an equation is represented as a single empty element with an external data entity pointing to a CGM file containing the equation.

If you want to represent a FrameMaker equation as an element with subelement structure instead, you must write a structure API client. If you do so, you will use the FDK object `FO_Math` and manipulate the equation structure as represented in MIF. For information on using the FDK to manipulate equations, see the *FDK Programmer's Guide*. For information on writing structure API clients, see the *Structure Import/Export API Programmer's Guide*.

Renaming markup attributes that correspond to graphic properties

FrameMaker represents properties of graphics (such as the height or width of the enclosing anchored frame) directly as part of the anchored frame, rather than as attributes on the graphic element. In markup however, these properties may be represented as attributes. In other words, there are certain markup attributes that do not translate to FrameMaker attributes, but instead directly to graphic properties. "Anchored frame properties" on page 302 and "Other graphic properties" on page 304 describe the default names for the markup attributes corresponding to graphic properties.

You can choose names other than the default ones for these markup attributes. To do so, use this version of the `attribute` rule:

```
attribute "attr" is fm property prop;
```

If your DTD uses a particular attribute name for the same purpose within multiple elements, you may want to use this rule as a highest-level rule to set a default. Otherwise, specify it within an `element` rule.

For example, assume you have four different markup elements representing graphics and equations. All four elements use the attribute h to represent the height of the anchored frame. Use the rule:

```
attribute "h" is fm property height;
```

With this rule, the software interprets the attribute h as the height of an anchored frame for all elements in which it occurs. If you use the same attribute for another purpose in another element, you'll have to write another `attribute` rule to handle it appropriately.

As another example, you may have only one element, `pic`, representing a graphic element and use the attribute h for unrelated purposes in other elements. In this case, you should restrict the association of h with the `height` property to the `pic` element instead of using the form of the rule that applies to all elements. You can use this rule:

```
element "pic" {
   is fm graphic element "Picture";
   attribute "h" is fm property height;
}
```

With this rule, other elements can use the h attribute for different purposes.

In both of these examples, the software creates an EDD from a DTD without creating an attribute that corresponds to the h attribute. When importing or exporting markup documents, it uses the attribute to read or write the appropriate information from the graphic.

The `dpi` and `impsize` attributes defined for graphic elements deserve special attention. For each generic identifier that represents an imported graphic object, you can decide whether its size is specified with the `dpi` or the `impsize` attribute. By default, the software uses the `dpi` attribute, but you can change this default with the `specify size` rule. For information on this rule, see "Changing how FrameMaker writes out the size of a graphic" on page 319.

For information on the rules used in these examples, see "attribute" on page 367, "element" on page 376, "is fm graphic element" on page 422, and "is fm property" on page 424.

Omitting representation of graphic properties in markup

Some properties of graphics and equations have no explicit representation in markup. In such cases, you may want to use a rule to make the information explicit in FrameMaker. You can use this rule to do so:

```
fm property prop value is "propval";
```

where *`prop`* is the FrameMaker property and *`propval`* is the value to use.

With this rule, the software creates a DTD from an EDD without creating the corresponding markup attribute. When importing a markup document, it causes the software to assign a particular value to one of the formatting properties. When exporting a document, it tells FrameMaker not to write an attribute with the value.

The `fm property` rule can be used at the highest level to set a default or within an `element` rule to be restricted to a single element.

For example, you may have a markup element `bitmap` treated as a graphic element in FrameMaker. Furthermore, you know that you never want to make a copy of such an object in the FrameMaker document; you do not want your end users to have the option of importing by copy. You can accomplish this as follows:

```
element "bitmap" {
   is fm graphic element;
   fm property import by reference or copy value is "ref";
}
```

When creating an EDD from a DTD, the software does not create an `impby` attribute for the `bitmap` element. Consequently, when exporting a FrameMaker document to markup, it does not try to write a value for the `impby` attribute. When importing a markup document, it automatically sets the property value to `ref`.

For a summary of the FrameMaker graphic properties, see "Text of default graphics and equations declarations" on page 300. For information on the rules used in this example, see "element" on page 376, "is fm graphic element" on page 422, and "fm property" on page 399.

Omitting optional elements and attributes from the default DTD declarations

The default DTD declarations provided with FrameMaker may be more general than your situation requires. If you create an initial version of your DTD from an EDD, it will always include the full set of default declarations for all graphic and equation elements. If you don't need all of this functionality, you may want to simplify the declarations.

For example, if your graphic elements are always created using FrameMaker's graphic tools, you could remove these declarations:

```
<!ATTLIST graphic_element
   dpi      NMTOKEN #IMPLIED <!--ignored if impsize specified-->
   impang   CDATA  #IMPLIED <!--import angle in frame in deg-->
   impby    (ref|copy) #IMPLIED <!--import by reference or copy-->
   impsize  CDATA  #IMPLIED <!--import size (width & height)-->
   sideways NMTOKEN #IMPLIED <!--1 if object turned sideways-->
   xoffset  CDATA  #IMPLIED <!--horizontal offset in frame-->
   yoffset  CDATA  #IMPLIED <!--vertical offset in frame-->
>
```

If you do so, FrameMaker won't produce or expect values for these attributes.

> **SGML:** In the above example, the `dpi` and `sideways` attributes are specified in XML using `NMTOKEN` for the numeric value. For SGML you can use `NUMBER`.

Specifying the data content notation on export

When the software writes an external data entity for a graphic or equation element, it uses the first eight characters of the facet name as the data content notation by default. If the graphic or equation element has only internal FrameMaker facets, it uses CGM as the data content notation. Your markup declarations may use different data content notations. If so, you can use the `notation` rule to associate a notation name with a facet or with general anchored frames or equations. The `notation` rule is of this form:

```
element "gi" {
   is fm graphic_or_equation element ["fmtag"];
   writer type ["name"] notation is "notation_name";
}
```

where *`gi`* is a generic identifier, *`graphic_or_equation`* is one of the keywords `graphic` or `equation`; *`fmtag`* is an optional FrameMaker element tag; *`type`* is one of the keywords `anchored frame`, `facet`, or `equation`; *`name`* is a facet name you supply only if *`type`*

is facet; and *notation_name* is the data content notation in the corresponding markup entity declaration.

For example, assume you have this rule:

```
element "graphic" {
   is fm graphic element;
   writer {
      facet "XWD" {
         notation is "xwd";
         export to file "$(docname).xwd";
}}}
```

In this case, when the software creates an external data entity for a Graphic element that has a facet whose name is xwd, it creates an entity declaration of this form:

```
<!ENTITY graphic1 SYSTEM "docname.xwd" NDATA xwd>
```

where docname is the name of the FrameMaker document. This example assumes that you have added an export filter to export a graphic in the XDump file format. For information on changing the filename written for this entity, see "Changing the name of the graphic file on export," next.

For more information on these rules, see:

- "notation is" on page 445
- "element" on page 376
- "is fm graphic element" on page 422
- "is fm equation element" on page 420
- "anchored frame" on page 365
- "equation" on page 385
- "facet" on page 394
- "writer" on page 467

Changing the name of the graphic file on export

Under certain conditions, when it exports a FrameMaker document, the software creates a file for each graphic and equation. For more information, see "Creating graphic files on export" on page 306. For the circumstances where the software will create a graphic file, you can change the name of the exported graphic file using the following rule:

```
element "gi" {
   is fm graphic_or_equation element ["fmtag"];
   writer type ["name"] export to file "fname";
}
```

where *gi* is a generic identifier; *graphic_or_equation* is one of the keywords graphic or equation; *fmtag* is an optional FrameMaker element tag; *type* is one of the keywords anchored frame, facet, or equation; *name* is a facet name you supply only if *type* is facet; and *fname* is the new filename. The *fname* argument can use these variables:

Variable	Meaning
$(entity)	The value of the corresponding markup element's entity attribute. If the source of the graphic inset wasn't originally a markup entity, this variable evaluates to a unique name based on the name of the element.
$(docname)	The name of the FrameMaker file, excluding any extension or directory information.

For example, assume you have the default declarations for the graphic element and you have this rule:

```
element "graphic" {
   is fm graphic element;
   attribute "entity" {
      is fm property entity;
   }
   writer facet "XWD" {
      notation is "xwd";
      convert referenced graphics;
      export to file "$(entity).xwd";
}}
```

With these rules, assume you imported a graphic element whose entity attribute had the value of flower. When you export the FrameMaker document to markup, the software writes this entity declaration:

```
<!ENTITY flower SYSTEM "flower.xwd" NDATA xwd>
```

It writes the graphic to a file named flower.xwd using the X Windows Dump format.

For more information on these rules, see:

- "export to file" on page 389
- "notation is" on page 445
- "attribute" on page 367
- "is fm property" on page 424
- "is fm attribute" on page 414
- "element" on page 376
- "is fm graphic element" on page 422
- "is fm equation element" on page 420

- “anchored frame” on page 365
- “equation” on page 385
- “facet” on page 394
- “writer” on page 467

Changing the file format of the graphic file on export

By default, when it creates a graphic file on export, FrameMaker writes the graphic file for a graphic or equation element in either CGM format or the format of the single facet it exports. For information on when the software creates a graphic file, see “Creating graphic files on export” on page 306.

You can tell the software to write graphics files in a different file format. To do so, you must first make sure that the format is one known to FrameMaker. For the list of graphic file formats FrameMaker supports, see “Supported graphic file formats” on page 298.

For information on which graphic export filters the software provides and on how to add new ones, see the online manual that describes using filters with FrameMaker products.

Once you are sure that the software can export your format of choice, you use a variant of the `export to file` rule described in “Changing the name of the graphic file on export,” (the previous section).

The general form of this rule is:

```
element "gi" {
   is fm graphic_or_equation element ["fmtag"];
   writer type ["name"] export to file "fname" as "format";
}
```

where *gi* is a generic identifier; *graphic_or_equation* is one of the keywords `graphic` or `equation`; *fmtag* is an optional FrameMaker element tag; *type* is one of the keywords `anchored frame`, `facet`, or `equation`; *name* is a facet name you supply only if *type* is `facet`; *fname* is the new filename; and *format* is the file format. The *fname* argument can use these variables:

Variable	Meaning
`$(entity)`	The value of the corresponding markup element’s `entity` attribute. If the source of the graphic inset wasn’t originally a markup entity, this variable evaluates to a unique name based on the name of the element.
`$(docname)`	The name of the FrameMaker file, excluding any extension or directory information.

For example, FrameMaker writes CGM files for all equation elements by default. If you want it to write QuickDraw PICT files instead, you can use this rule:

```
element "eqn" writer equation
   export to file "eqn.pic" as "PICT";
```

For more information on these rules, see "export to file" on page 389, "element" on page 376, "is fm graphic element" on page 422, "is fm equation element" on page 420, "anchored frame" on page 365, "equation" on page 385, "facet" on page 394, and "writer" on page 467.

Changing the file format for graphics imported by reference

By default, if a graphic or equation element contains a single graphic file that is imported by reference, when exporting a FrameMaker document the software does not create a new graphic file for the element. However, you can force the software to create a new graphic file for such elements. To do this, you use the `convert referenced graphics` rule. Note that this rule can only be used as a subrule of a `facet` rule.

For example, assume you want to convert all graphic files to the PICT format. With the following example, the software would create PICT files for every graphic:

```
element "graphic" {
   is fm graphic element;
   writer facet default {
      convert referenced graphics;
      export to file "$(entity).pic" as
         "PICT";
}}
```

Depending on how a graphic element was created in your FrameMaker document, it would export as follows:

For a graphic element:	On export to markup the software:
Imported as a markup graphic element that used the `entity` attribute to refer to an external data entity named *ename* (where *ename* is the name of the entity)...	Writes a graphic file named *ename*`.pic`. It also creates a graphic element in markup with *ename* as the value of the `entity` attribute, and an entity named *ename* that references the graphic file.
Imported as an SGML graphic element that used the `file` attribute to refer to a graphic file named *fname* (where *fname* is the name of the graphic file)...	Writes a graphic file named *fname*`.pic`. It also creates an SGML graphic element with *fname* as the value of the `file` attribute.

For a graphic element:	On export to markup the software:
Created in the FrameMaker document by the author...	Writes a graphic file named `graphic1.pic` (`graphic2.pic`, `graphic3.pic`, etc.). It also creates a graphic element in markup with `graphic1` as the value of the `entity` attribute, and an entity named `graphic1` that references the graphic file.

You can use a similar rule to convert all graphic files of one format to another. With the following example you can convert all TIFF files to PICT:

```
element "graphic" {
   is fm graphic element;
   writer facet "TIFF"{
      convert referenced graphics;
      export to file "$(entity).pic" as
         "PICT";
}}
```

Specifying the entity name on export

Your organization may have common graphics used by all authors. For example, you may have a particular graphic file that contains your company's logo. To facilitate authors using the same graphic for the logo, you can create an element that always points to the same file. In this case, you always want the entity name to be the same for elements of this type.

In the absence of an entity property value for the graphic inset, when the software exports an external data entity for a graphic or equation, it generates a name for the entity based on the element name. You can change the name with the `entity name` rule. The format of the `entity name` rule is:

```
element "gi" {
   is fm graphic_or_equation element ["fmtag"];
   writer type ["name"] entity name is "ename";
}
```

where *gi* is a generic identifier; *graphic_or_equation* is one of the keywords `graphic` or `equation`; *fmtag* is an optional FrameMaker element tag; *type* is one of the keywords `anchored frame`, `facet`, or `equation`; *name* is a facet name you supply only if *type* is `facet`; and *ename* is the entity name. In situations where there could be more than one entity for a given graphic element, the software ensures unique entity names by appending an integer to the end of the *ename* argument.

Assume authors import a graphic by reference into a FrameMaker element named Logo. Also assume the graphic is in a file named `cologo.tif` in TIFF format, so you could use the rule:

```
element "logo" {
   is fm graphic element;
   writer facet "TIFF" {
      entity name is "cologo";
}}
```

With this rule, the software creates a single instance of the following entity declaration in the markup document's internal DTD subset:

```
<!ENTITY cologo1 SYSTEM cologo1.tif NDATA TIFF>
```

For more information, see "Creating graphic files on export" on page 306. For information on specifying a graphic filename, see "Changing the name of the graphic file on export" on page 314. For more information on these rules, see:

- "entity name is" on page 383
- "element" on page 376
- "is fm graphic element" on page 422
- "is fm equation element" on page 420
- "anchored frame" on page 365
- "equation" on page 385
- "facet" on page 394
- "writer" on page 467

Changing how FrameMaker writes out the size of a graphic

The default declarations for graphics include both `dpi` and `impsize` attributes with the software using one or the other of these attributes when it exports a single graphic facet. However, you can override the software's default behavior by using the `specify size` rule.

This rule determines which of these attributes the software writes. In addition, it indicates what units to use for `impsize`. If there is no `specify size` rule, FrameMaker uses the `dpi` attribute. In all cases it uses a resolution of 0.001 (to 3 decimal points). The general form of this rule is:

```
element "gi" {
   is fm graphic_or_equation element ["fmtag"];
   writer type ["name"]
      specify size in units;
}
```

where *gi* is a generic identifier; *graphic_or_equation* is one of the keywords graphic or equation; *fmtag* is an optional FrameMaker element tag; *type* is one of the keywords anchored frame, facet, or equation; *name* is a facet name you supply only if *type* is facet; *units* indicates the unit of measure in for the graphic.

FrameMaker reports the size of the imported graphics object in the indicated units. It calculates a value of impsize by determining the width and height of the smallest possible rectangle that can contain the imported object using the default resolution and unit type.

For example, assume the graphic is a circle with a diameter of 3.1115 centimeters. Given the rule:

```
specify size in cm;
```

FrameMaker generates the attribute impsize="3.112cm 3.112cm".

For more information on these rules, see:

- “specify size in” on page 456
- “element” on page 376
- “is fm graphic element” on page 422
- “is fm equation element” on page 420
- “anchored frame” on page 365
- “equation” on page 385
- “facet” on page 394
- “writer” on page 467

20 Translating Cross-References

A *cross-reference* is a passage in one place in a document that refers to another place, a *source*, in the same document or a different document. While markup does not explicitly support cross-references, it does provide the declared values `ID`, `IDREF`, and `IDREFS` for attributes; and attributes using these declared values customarily represent cross-references. FrameMaker can also use this model for cross-references within a FrameMaker document.

There are several differences between the FrameMaker cross-reference mechanism and the customary way of interpreting the related markup attributes. For information on these differences, see "Cross-references" on page 21.

In this chapter

This chapter starts by describing the default translation provided by FrameMaker. What FrameMaker does by default differs depending on whether you start from an EDD or from a DTD and whether you are working with SGML or XML. The chapter then describes how to modify the default behavior. In the outline below, click a topic to go to its page.

How FrameMaker translates cross-references by default:

- "On export to markup" on page 322
- "On import to FrameMaker" on page 323

Some ways you can change the default translation:

- "Translating markup elements as FrameMaker cross-reference elements" on page 324
- "Renaming the markup attributes used with cross-references" on page 325
- "Translating FrameMaker cross-reference elements to text in markup" on page 326
- "Maintaining attribute values with FrameMaker" on page 326
- "Translating external cross-references to and from XML" on page 326

Default translation

On both import and export, FrameMaker assumes that elements having an attribute with a declared value of `ID` can be the source of a cross-reference. It also assumes markup language elements with declared content `EMPTY` that have an attribute with a declared value of `IDREF` are for cross-references. So, by default, when processing a single markup document, FrameMaker document, or FrameMaker book, FrameMaker automatically

translates EMPTY markup language elements that use an IDREF attribute to or from FrameMaker cross-reference elements.

FrameMaker interprets internal and external cross-references in multiple-file markup documents and FrameMaker books in special ways:

- When you export a FrameMaker *book* to markup, it becomes a single document instance in markup. Similarly, when you import a markup *document* to a FrameMaker book, it becomes multiple FrameMaker documents. Because of this, cross-references that are external in FrameMaker may be internal to a single document instance. For a discussion of the importance of the distinction between internal and external cross-references, see "Cross-references" on page 21.
- When you export a *single* FrameMaker document to markup, FrameMaker treats all cross-references that were external in FrameMaker as external in markup. However, if you export an *entire* FrameMaker book to markup, treatment of the individual FrameMaker documents that make up the book is more complicated: cross-references between FrameMaker documents in the book are considered internal cross-references in the markup—only those cross-references to documents not in the FrameMaker book are considered external.

For example, assume your FrameMaker book, manual.book, contains the files ch1.fm and ch2.fm and that there is an external cross-reference in ch1.fm to ch2.fm. When you export manual.book to markup, it creates one document instance and the external cross-reference from ch1.fm to ch2.fm becomes an internal cross-reference in this document instance. If you import this markup document back into FrameMaker (thus again creating multiple files), the cross-reference again becomes an external cross-reference. For information on exporting FrameMaker books, see Chapter 24, "Processing Multiple Files as Books."

On export to markup

When creating a DTD from an EDD, FrameMaker translates ID attributes used for cross-reference sources in the same way as any other attributes. FrameMaker translates a cross-reference element as an empty markup element of the same name. The element's attributes are exported as well. To simplify export, your cross-reference element should have an IDREF attribute.

When creating a DTD, for each cross-reference element, FrameMaker creates an additional impliable attribute, format, with the declared value CDATA. When exporting a FrameMaker document, the value of this attribute is set to the name of the cross-reference format used by the cross-reference. This attribute corresponds to the property cross-reference format in read/write rules. Furthermore, in XML DTDs, FrameMaker also creates an impliable CDATA attribute called srcfile. This attribute is used for external cross-references and cannot be modified by read/write rules.

When exporting a document, FrameMaker exports ID and IDREF attributes as it does any other attributes. Thus, if you have used an element for the source of your cross-reference

and an IDREF attribute on the cross-reference element, the export process works simply to provide a natural markup representation.

If you define a cross-reference element without an IDREF attribute, then the software uses an internal mechanism for storing cross-reference information. It does not export this information to markup.

Whether or not the cross-reference element has an IDREF attribute, if the source of the cross-reference is not an element, then FrameMaker exports the cross-reference element as text. That is, the fact that an element was present in the FrameMaker document is lost; instead, the text content provided by the cross-reference format appears in the markup document.

If the cross-reference source is external to the file when exporting a single file or external to the book when exporting a book, then FrameMaker exports the cross-reference to SGML as text. It exports such a cross-reference to XML as a cross-reference element, setting its srcfile attribute to the URI of the file containing the source element with the ID of the source element as a fragment identifier. In other words, the value of the srcfile attribute is the URI of the source file followed by a # delimiter and the value of the ID attribute. See "Translating external cross-references to and from XML" on page 326 for examples.

A cross-reference source that isn't an element is exported in the same way as any other FrameMaker non-element marker. Its corresponding cross-reference exports as text in the way described above.

For more information on FrameMaker's representation of cross-references, see "Using UniqueID and IDReference attributes" on page 176 of this manual and see the FrameMaker and FrameMaker user's manuals.

On import to FrameMaker

When creating an EDD from a DTD, FrameMaker creates a cross-reference element when it encounters an element declaration with declared content EMPTY that has an attribute with the declared value IDREF or IDREFS. When it encounters any other element declaration with an attribute with declared value IDREF or IDREFS, it creates an element and treats the attribute as an ordinary attribute. When it encounters an attribute with the declared content ID, it creates a FrameMaker UniqueID attribute.

When importing a markup document, FrameMaker creates the appropriate attribute in FrameMaker if it encounters an attribute with the declared value ID. When it encounters an element with a single attribute with the declared value IDREF and the declared content EMPTY, it translates the element to a cross-reference element. If your markup document is invalid in that it does not include both the IDREF attribute and its corresponding ID attribute, this process results in an unresolved cross-reference. If your XML document also defines a srcfile attribute for such an element, FrameMaker interprets its value as that of the URI of the cross-reference source.

When importing a markup document, FrameMaker creates a cross-reference element when it encounters an element with a single attribute whose declared value is IDREFS and declared content is EMPTY. The software uses the first value in the IDREFS attribute as the ID of the cross-reference. It saves the other values, to write out on export, but does nothing special with them. You must write a structure API client to change this behavior.

Other markup elements may be intended to represent cross-references. For example:

- SGML elements that have attributes with the declared values NAMES
- Elements with an IDREFS attribute
- Elements that have multiple attributes with the declared value IDREF
- Elements that are not EMPTY yet have attributes with the declared value

You can write a structure API client to handle these situations.

Modifications to the default translation

The following sections describe some of the possible modifications to the default translation of cross-references. You may require different rules or a structure API client, or you may use these rules in ways not discussed in these sections. For additional ways to modify the translation of cross-references, see the information at the end of each section.

For a summary of read/write rules relevant to translating cross-references, see "Cross-references" on page 356. For information on writing structure API clients, see *Structure Import/Export API Programmer's Guide*.

Translating markup elements as FrameMaker cross-reference elements

You can identify which markup elements correspond to FrameMaker cross-reference elements and choose to use the same name or different names for the markup and FrameMaker elements. The rules for doing this apply on both import and export to create elements of the appropriate type. In general, you use a rule of this form:

```
element "gi" is fm cross-reference element fmtag;
```

where *gi* is a generic identifier and *fmtag*, if specified, indicates the name of the corresponding FrameMaker element. For example, to indicate that both of the markup elements xref and link correspond to FrameMaker cross-reference elements, you can use these rules:

```
element "xref" is fm cross-reference element "XRef";
element "link" is fm cross-reference element;
```

For information on the rules used in this example, see "element" on page 376 and "is fm cross-reference element" on page 418.

Renaming the markup attributes used with cross-references

(For details, see “On export to markup” on page 322) FrameMaker creates the `format` attribute for working with cross-references but you can choose a different attribute for the same purpose. You can do so either at the highest level to set a default or within an `element` rule for a specific markup element. Use the `is fm property` rule within an `attribute` rule either at the highest level or within an `element` rule. That is, use a rule of one of these forms:

```
attribute "attr"
   is fm property cross-reference format;

element "gi"
   attribute "attr"
      is fm property cross-reference format;
```

where *gi* is a generic identifier and *attr* is a markup attribute.

For example, to specify that all relevant markup elements use the `fmform` attribute to specify a FrameMaker cross-reference format, but that the markup `exref` element uses the `exform` attribute, you would use these rules:

```
attribute "fmform" is fm property cross-reference format;

element "exref" {
   is fm cross-reference element "CrossRef";
   attribute "exform" is fm property cross-reference format;
}
```

Instead of translating the cross-reference format as an attribute, you may choose to use the `fm property` rule to explicitly set the property value. You can use the `fm property` rule for this purpose.

You can use the `is fm property rule` to specify an attribute to use as a cross-reference Id. For example, if the attribute `linkend` stores the `IDREF` for a markup element, you can set that value to be the FrameMaker cross-reference ID property with the following rule:

```
attribute "linkend" is fm property cross-reference id;
```

On export, instead of writing the cross-reference Id to the `IDREF` attribute of the element, the software will write that value to the `linkend` attribute.

For information on the rules used in this example, see “element” on page 376, “attribute” on page 367, “is fm property” on page 424, and “is fm cross-reference element” on page 418.

Translating FrameMaker cross-reference elements to text in markup

You may not want your FrameMaker cross-reference elements to remain elements in the markup representation—you can choose to translate them to text instead. To do so, use this rule:

```
fm element "fmtag" unwrap;
```

where *fmtag* is the FrameMaker element tag for a cross-reference element.

For more information on these rules, see "fm element" on page 397 and "unwrap" on page 461.

Maintaining attribute values with FrameMaker

FrameMaker can maintain values for `ID` and `IDREF` attributes so that your end users do not need to keep track of the values. If you want to let the software do so, you may choose also to prohibit your end users from changing the values of these attributes.

For information on how to maintain this control, see "Creating read-only attributes" on page 244.

Translating external cross-references to and from XML

The source of an internal cross-reference is completely identified by the value of its `ID` attribute. Hence, internal cross-references are specified with `IDREF` attributes. To identify the source of an external cross-reference, though, the file name of the containing document is also needed. As explained on page 323, when FrameMaker saves a document as SGML, it avoids the need to export a file name by converting external cross-references to text.

When FrameMaker saves a structured document to XML, it preserves cross-reference elements for external cross-references, using the `srcfile` attribute to export the source's file name. Within FrameMaker, an external cross-reference links one FrameMaker document to another FrameMaker document. An exported cross-reference can refer either to the FrameMaker document that was the original cross-reference source, or to a corresponding XML document. For example, suppose the source of an external cross-reference is an element with its `ID attribute set to BAAHDJJE` that occurs in the file `c:\myproject\manual\intro.fm`. If the name of the cross-reference element is `xref` and the cross-reference format is not exported, by default FrameMaker exports the cross-reference as:

```
<xref srcfile="file:///C:/myproject/manual/intro.fm#BAAHDJJE">
```

If you save the file containing the cross-reference source to XML, you can choose instead to set the `srcfile` attribute to refer to the XML version of that file. You specify this option in the application definition in `structapps.fm` or another application definition file rather than with read/write rules. Insert an `ExternalXRef` child in the `XMLApplication` element for the application you are using. In this `ExternalXRef` element, insert a `ChangeReferenceToXML` child. Finally, insert an `Enable` element into the `ChangeReferenceToXML` element. Make sure you restart FrameMaker or use the File >

Structure Tools > Read Application Definitions command to process the modified application definition. When you save the document with the cross-reference, FrameMaker then changes the extension in the `srcfile` attribute to `.xml` and exports the cross-reference as:

```
<xref srcfile="file:///C:/myproject/manual/intro.xml#BAAHDJJE">
```

You can save the source file to XML before or after saving the original file to XML. In either case, make sure that the file name you specify for the XML document differs from that of the original FrameMaker document only in the extension.

If you insert a `Disable` element instead of an `Enable` element into `ChangeReferenceToXML`, FrameMaker retains the default behavior and does not change the extension in the `srcfile` attribute.

The `ExternalXRef` element in the application definition file has another possible child. `TryAlternativeExtensions` specifies an option for importing external cross-references from XML. If its content is `Enable`, if FrameMaker cannot open the file specified by the `srcfile` attribute, it changes its extension and tries to open the resulting file instead. In particular, if the original extension is `.xml`, FrameMaker also tries `.fm`; if the original extension is `.fm`, FrameMaker also tries `.xml`. If the content of `TryAlternativeExtensions` is `Disable`, FrameMaker creates an unresolved cross-reference if the specified file cannot be opened. `Disable` is the default.

21 Translating Variables and System Variable Elements

You use variables in FrameMaker documents to store information that may change at some later time, such as a product's name; information you know will change, such as the current date; or text that you must enter frequently. Variables make it easier for you to manage these changes.

In markup, you can use either elements or entities for similar purposes. Some of the material in this chapter is closely related to the handling of entities. For more information on entities, see Chapter 17, "Translating Entities and Processing Instructions."

In this chapter

This chapter starts by describing FrameMaker's default translation of variables. The chapter then describes modifications you can make to the default behavior. Some of these procedures are relevant when translating in both directions; others are relevant only in one direction. In the outline below, click a topic to go to its page.

How FrameMaker translates variables by default:

- "On export to markup" on page 330
- "On import to FrameMaker" on page 331

Some ways you can change the default translation:

- "Renaming or changing the type of entities when translating to variables" on page 332
- "Translating markup elements as system variable elements" on page 333
- "Translating FrameMaker system variable elements to text in markup" on page 333
- "Translating FrameMaker variables as SDATA entities" on page 334
- "Discarding FrameMaker variables" on page 334

Default translation

Markup has no unique representation for variables. There are two types of FrameMaker variables:

- User-defined variables provide an easy way to store information that may change. For example, in an insurance policy document, you might represent the name of the insured person with the variable `Insured`, which FrameMaker replaces with the name of the insured person for a particular policy.

- System variables are used to insert system specific calculations, such as the current date or the current page number. You cannot modify system variables.

In addition, FrameMaker provides system variable elements. These elements are reflected in the structure of the document. You use them for the same purposes as system variables.

For more information on FrameMaker variables and system variable elements, see the FrameMaker user's manual.

On export to markup

FrameMaker translates a system variable element as an empty markup element. It does not record the definition of the corresponding variable, because that is formatting information for the element.

When writing a markup document, FrameMaker bases its treatment of non-element variables on the variable text. Unless instructed otherwise, it translates a user variable to a reference to an entity with the same name as the variable. If no entity definition with the given name exists, FrameMaker creates a new entity definition using the variable name and variable text. If an entity definition with the given name does exist, FrameMaker writes a message to the log file warning of a potential mismatch.

If the variable name is not a valid name in markup, FrameMaker uses `fmv1` as a default entity name. If an entity already exists by that name, FrameMaker increments the counter until an unused name is found, for example, `fmv2`, `fmv3`, and so forth.

FrameMaker determines the type of entity on the basis of special character formats within the variable text. If the variable text uses the `FmCdata` character format, FrameMaker exports the variables as a `CDATA` entity. If the variable text uses the `FmSdata` character format, FrameMaker exports it as an `SDATA` entity. In the absence of any relevant character format information and markup, FrameMaker exports the variable as a text entity. If the variable text contains markup, then FrameMaker exports the variable as a `CDATA` entity.

XML: The XML standard does not allow `SDATA` and `CDATA` entities. If your document contains variables that would translate to either of these entity types, FrameMaker exports them as text, and the use of the variable will be lost the next time you import the XML data.

To retain the use of variables, you should map FrameMaker variables to specific XML elements. (See "Translating markup elements as system variable elements" on page 333.) Also, you should never include markup in variables if you intend to export your document to XML.

SGML: FrameMaker reports special characters within `CDATA` entities as errors. For SGML text, external data, and `CDATA` entities, the entity text is the same as the variable text. For `SDATA` entities, FrameMaker uses the string `"FM variable"` as the parameter literal.

FrameMaker translates a non-element system variable as a reference to an entity in SGML with one of the following names:

System variable	Entity
Page Count	`fm.pgcnt`
Current Date (Long)	`fm.ldate`
Current Date (Short)	`fm.sdate`
Creation Date (Long)	`fm.lcdat`
Creation Date (Short)	`fm.scdat`
Modification Date (Long)	`fm.lmdat`
Modification Date (Short)	`fm.smdat`
Filename (Long)	`fm.lfnam`
Filename (Short)	`fm.sfnam`
Table Continuation	`fm.tcont`
Table Sheet	`fm.tsht`

For more information on FrameMaker's treatment of entities, see Chapter 17, "Translating Entities and Processing Instructions."

On import to FrameMaker

By default, FrameMaker does not create system variable elements when creating an EDD from a DTD. It creates user variables for entities of various sorts. For information on how FrameMaker translates entities by default as variables, see "On import to FrameMaker" on page 251.

Modifications to the default translation

The following sections describe some possible modifications to the default translation of FrameMaker variables and system variable elements. You may require different rules or a structure API client, or you may use these rules in ways not discussed in these sections. For additional ways to modify the translation of variables and system variable elements, see the cross-references at the end of each procedure and in Chapter 17, "Translating Entities and Processing Instructions."

For a summary of read/write rules relevant to translating variables and system variable elements, see "Variables" on page 362 and "Entities" on page 357. For information on writing structure API clients, see the *Structure Import/Export API Programmer's Guide*.

Renaming or changing the type of entities when translating to variables

By default, FrameMaker converts non-element system variables and user variables to entity references. You can change the name of the entity to use for a variable with the `entity` rule. The format of this rule is:

```
entity "ename" is fm variable "fmvar";
```

where *ename* is a markup entity and *fmvar* is a FrameMaker variable. With this rule, FrameMaker creates a reference to an entity of the appropriate type, on the basis of character formats associated with the variable text. It also creates an entity declaration if one doesn't already appear in the internal DTD subset. For information on how character formats affect the translation of variables on export, see "On export to markup" on page 330.

You can change the type of entity to which a variable translates by changing the character format associated with the variable text in the variable definition.

For example, assume you have the FrameMaker variable `Start Tag Ex` with the variable text <tag> and that you want to translate it to the SGML CDATA entity `startex`. To do so, use this rule:

```
entity "startex" is fm variable "Start Tag Ex";
```

Be sure that `Start Tag Ex` uses the CDATA character format in its definition. That is, the definition of the variable must be:

```
<CDATA>\<tag\><Default ¶ Font>
```

If `startex` is not declared in the DTD, FrameMaker inserts the following entity declaration into the DTD subset when it encounters the first instance of the variable `Start Tag Ex` in the document being processed:

```
<!ENTITY startex CDATA "<tag>">
```

SGML: If you want the variable to become an SDATA entity instead, change the variable definition to:

```
<SDATA>\<tag\><Default ¶ Font>
```

In this case, if `startex` is not declared in the DTD, FrameMaker inserts this entity declaration:

```
<!ENTITY startex SDATA "<tag>">
```

If the DTD already has an appropriate entity declaration, FrameMaker retains that declaration.

In a similar circumstance, assume you map a text entity named `product` to a FrameMaker variable, `Product Name`. If you have the following text entity declaration:

```
<!ENTITY product "Not yet named">
```

then with the rule:

```
entity "product" is fm variable "Product Name";
```

FrameMaker produces a reference to the general entity, but doesn't create a new entity declaration.

For information on the rules used in these examples, see "entity" on page 381 and "is fm variable" on page 441.

Translating markup elements as system variable elements

You can translate some markup elements as FrameMaker system variable elements. To do so, use the following rule:

```
element "gi" is fm system variable element ["fmtag"];
```

where *gi* is a generic identifier and the optional argument *fmtag* is a FrameMaker element tag corresponding to a system variable element. For example, to translate a markup element `date` as a system variable element of the same name, you could use this rule:

```
element "date" is fm system variable element;
```

To rename the element on import and export, you could use this rule:

```
element "date" is fm system variable element "TodaysDate";
```

This rule translates the markup element `date` as the FrameMaker system variable element `TodaysDate`. You cannot use a read/write rule to specify which system variable to associate with the element, because this association is considered formatting information. Instead, you must add appropriate format rules to the EDD. For example, you could have this definition in your EDD:

Element (System Variable): TodaysDate
System variable format rule
1. In all contexts.
Use system variable: Current Date (Long)

In this case, `TodaysDate` always translates to the system variable Current Date (Long).

For information on these rules, see "element" on page 376 and "is fm system variable element" on page 433.

Translating FrameMaker system variable elements to text in markup

You may not want your FrameMaker system variable elements to remain elements in the markup representation. You can choose to translate them to text in markup instead. To do so, use this rule:

```
fm element "fmtag" unwrap;
```

where *fmtag* is the FrameMaker element tag for a system variable element.

For more information on these rules, see "fm element" on page 397 and "unwrap" on page 461.

Translating FrameMaker variables as SDATA entities

For SGML, you can translate FrameMaker variables as SDATA entities by using the `entity` rule or by manipulating parameter literals. For information on how to do so, see "Translating entities as FrameMaker variables" on page 259.

Discarding FrameMaker variables

FrameMaker always allows you to insert a variable in a document. To modify this behavior, you must use an FDK client. If you don't want FrameMaker to export some or all variables to markup, you can choose to have it discard all variables or particular variables.

To have FrameMaker discard variables, use this rule:

```
fm variable ["var1", . . ., "varn"] drop;
```

Each var_i is a variable. If you don't specify var_i in the rule, the rule applies to all variables not addressed explicitly by `entity` or other `fm variable` rules. It is an error if the same var_i appears in multiple `entity` or `fm variable` rules.

This rule always occurs in a highest-level rule, because it applies to all instances of the indicated variables. It does not apply to system variable elements.

For information on the rules used in this example, see "writer" on page 467, "fm variable" on page 402, and "drop" on page 373.

22 Translating Markers

You use markers in FrameMaker documents to store various kinds of information you don't want visible to the document's audience. Although there's no analogous concept in markup, markers frequently correspond to various attributes or elements.

FrameMaker provides a variety of marker types. Two of these marker types have special default translations.

- DOC PI markers store information about some processing instructions, and DOC Entity Reference markers store information about entity references. For information on this use of markers, see Chapter 17, "Translating Entities and Processing Instructions." You can change which marker type FrameMaker uses to store this information.
- Conditional Text markers indicate conditional text. FrameMaker ignores conditional text markers on export to markup. For information on the treatment of conditional text on export, see "Marked sections and conditional text" on page 23.

In this chapter

This chapter describes the default translation of markers and modifications you can make to the default behavior. Some of these procedures are relevant when translating in both directions; others are relevant only in one direction.

How FrameMaker translates markers by default:

- "On export to markup" on page 336
- "On import to FrameMaker" on page 336

Some ways you can change the default translation:

- "Translating markup elements as FrameMaker marker elements" on page 337
- "Writing marker text as element content instead of as an attribute" on page 337
- "Using markup attributes and FrameMaker properties to identify markers" on page 338
- "Discarding non-element FrameMaker markers" on page 339

Default translation

Markup has no special representation for markers. FrameMaker allows you to represent markers either as marker elements or as non-element markers.

On export to markup

FrameMaker exports a marker element as an markup empty element of the same name, with two additional attributes, `text` and `type`. The value of the `text` attribute is the marker text; the value of the `type` attribute is the marker type.

> **XML and SGML:** The XML specification indicates that closing delimiter string for a PI is ?>. Unless otherwise stated, the examples of PIs in this section use the XML syntax.

FrameMaker exports non-element markers, other than those of Type `DOC PI`, as processing instructions of the following form:

```
<?FM MARKER [type] text?>
```

where *type* is the marker type and *text* is the marker text. For example, FrameMaker exports an Index marker with the text “translating, PIs” as:

```
<?FM MARKER [Index] translating, PIs?>
```

If a DOC PI marker has marker text of this form:

```
text
```

then FrameMaker outputs this processing instruction:

```
<?text?>
```

If a DOC Entity Reference marker has marker text of this form:

```
entname
```

where *entname* is the entity name, then FrameMaker outputs this entity reference:

```
&entname;
```

You can use a rule to change the marker type whose text is treated in this manner.

FrameMaker uses the Cross-Ref marker type to mark the source of a non-element cross-reference. On export to markup, these markers are treated as any other non-element markers are. They are not treated specially to indicate the source of a cross-reference. For information on exporting cross-references, see Chapter 20, “Translating Cross-References.”

On import to FrameMaker

In the absence of read/write rules, FrameMaker cannot identify a markup element that corresponds to a marker. If you have elements you want to translate as markers, you must write rules.

However, if you start with an EDD instead of a DTD, the default DTD translates FrameMaker marker elements as markup elements with a declared content of `EMPTY`. If you have a markup document that uses this DTD, and your application specifies a FrameMaker

template to use on import, then it translates the appropriate markup elements to marker elements when you import the document to FrameMaker.

Also, by default FrameMaker imports some entity references and processing instructions as non-element markers. For information on when this happens, see Chapter 17, "Translating Entities and Processing Instructions."

Modifications to the default translation

The following sections describe some modifications to the default translation of markers. You may require different rules or a structure API client, or you may use these rules in ways not discussed in these sections. For additional ways to modify the translation of markers, see the cross-references at the end of each procedure.

For a summary of read/write rules relevant to translating markers, see "Markers" on page 359.

Translating markup elements as FrameMaker marker elements

You can identify which markup elements correspond to FrameMaker marker elements, and use the same name or different names for the FrameMaker and markup elements. This approach works for creating elements of the appropriate type on both import and export. In general, you use a rule of this form:

```
element "gi" is fm marker element ["fmtag"];
```

where *gi* is a generic identifier and *fmtag* is a FrameMaker element tag. If *fmtag* is specified, it is the name of the corresponding FrameMaker marker element; otherwise, the FrameMaker element name is the same as the generic identifier.

For information on the rules used in this example, see "element" on page 376 and "is fm marker element" on page 423.

Writing marker text as element content instead of as an attribute

By default, on export FrameMaker writes the text of a marker element as the value of an attribute of an empty markup element. On import, it finds the marker text in that attribute. Instead, you can choose to have the marker text stored as the content of the corresponding markup element. Note that the markup element cannot be declared as an empty element.

To treat marker text for a specific FrameMaker marker element as content for the markup element, use the `marker text` rule as a subrule of the `element` rule:

```
element "gi" marker text is content;
```

where *gi* is a generic identifier.

For information on the rules used in these examples, see "element" on page 376 and "marker text is" on page 444.

Using markup attributes and FrameMaker properties to identify markers

When translating FrameMaker marker elements to markup elements, by default FrameMaker uses the markup attributes `type` and `text`. These correspond, respectively, to the FrameMaker properties `marker type` and `marker text`. You can choose to have FrameMaker use different attribute names or to not use one or both of the attributes altogether.

To have FrameMaker not use one of these attributes, remove the attribute from the definition of the corresponding markup element in the DTD, or drop the attribute on import. When exporting a FrameMaker document, if the corresponding markup element does not have one of these attributes defined, the software does not write a value for the attribute.

To rename these properties or attributes, use the following rule:

```
element "gi" attribute "attr" is fm property prop;
```

where *gi* is a generic identifier, *attr* is a markup attribute and *prop* is one of `marker type` or `marker text`. For example, to have the markup element `index` become a marker element of type Index and get its text from the `term` attribute, you could use the following rule:

```
element "index" {
   is fm marker element;
   attribute "term" is fm property marker text;
}
```

and this element definition:

Element (Marker): Index
Initial marker type
1. In all contexts.
Use marker type: Index

With this rule and declaration, FrameMaker translates an `Index` element to a start-tag of the form:

```
<index term="Some index text">
```

For information on the rules used in this example, see:

- “element” on page 376
- “is fm marker element” on page 423
- “fm property” on page 399
- “drop” on page 373
- “attribute” on page 367
- “is fm property” on page 424

Discarding non-element FrameMaker markers

FrameMaker always allows you to insert a non-element marker in a document. Modifying this behavior requires an FDK client. However, if you don't want FrameMaker to export some or all non-element markers to markup, you can choose to have it discard either all non-element markers or non-element markers of certain types.

To discard non-element markers on export of a FrameMaker document, use this rule:

```
fm marker ["type1", . . ., "typen"] drop;
```

Each *type1* is a marker type. A particular *type1* can appear in only one rule. If there is a rule with no *type1* specified, it provides a default for marker types not explicitly listed in another rule.

You may choose to drop most non-element markers and retain only certain types as processing instructions. In this case, you could write rules to translate individual types as processing instructions, followed by a rule to drop all non-element markers. For example, to drop all non-element markers except Index and Hypertext markers, use these rules:

```
fm marker "Index" is processing instruction;
fm marker "Hypertext" is processing instruction;
fm marker drop;
```

Since FrameMaker uses the first rule that matches in any given situation, the order of these rules is important. If the rules occurred in this order:

```
fm marker drop;
fm marker "Index" is processing instruction;
fm marker "Hypertext" is processing instruction;
```

then FrameMaker would drop all non-element markers, including Index and Hypertext markers.

For information on the rules used in this example, see "fm marker" on page 398, "drop" on page 373, and "is processing instruction" on page 441. For information on writing FDK clients, see *FDK Programmer's Reference*, an online manual supplied with the Frame Developer's Kit.

23 Translating Conditional Text

Conditional text is a FrameMaker mechanism for specifying portions of a document that can be included or omitted as needed. For information on working with conditional text, see the *FrameMaker User Guide*.

When FrameMaker exports a structured document to SGML, it exports shown conditional text, but does not indicate that the text was conditional. It does not export hidden conditional text.

However, FrameMaker can represent both condition settings and the application of condition tags in XML in a way that permits round-tripping of this information. The remainder of this chapter therefore applies only to XML.

In this chapter

This chapter starts by describing the default translation provided by FrameMaker. The chapter then describes how to modify the default behavior. In the outline below, click a topic to go to its page.

How FrameMaker translates conditional text by default:

- "On export to markup" on page 343
- "On import to FrameMaker" on page 344

How you can change the default translation:

- "Modifications to the default translation" on page 344

Default translation

To preserve conditional text in XML, two types of information are needed:

- Condition settings, including the condition tags defined in the FrameMaker document as well as the condition indicators associated with each and whether the condition is shown or hidden
- Condition tags, if any, associated with each portion of the document's content

Neither type of information is stored in elements in the FrameMaker document. Condition settings are independent of the document's element structure and one or more condition tags can be applied to an element or to all or part of its content. FrameMaker therefore uses processing instructions to represent conditional text in XML.

Condition settings

A separate processing instruction at the beginning of the XML document defines each condition tag. This processing instruction has the form:

```
<?Fm Condition tag color style status?>
```

where

- *tag* is a condition tag
- *color* is the condition indicator color (AsIs if no color is specified)
- *style* is the condition indicator style, represented by one of the following keywords:

Keyword	Condition Indicator (as defined in the Style pop-up menu of the Edit Condition Tag dialog box)
NO_OVERRIDE	As Is
OVERLINE	Overline
STRIKETHROUGH	Strikethrough
SINGLE_UNDERLINE	Underline
DOUBLE_UNDERLINE	Double Underline
CHANGEBAR	Change Bar
NUMERIC_UNDERLINE	Numeric Underline
NMRIC_AND_CHNGBAR	Numeric Underline and Change Bar

- *status* is one of the keywords hide or show to indicate the show/hide status of the condition tag

For example, the following processing instructions define conditions tagged Summer and Winter:

```
<?Fm Condition Summer Blue OVERLINE hide?>
<?Fm Condition Winter Red SINGLE_UNDERLINE show?>
```

Conditional text

A conditional portion of an XML document's content is delimited by processing instructions that identify the applied condition tag. The processing instruction that precedes the conditional text has the form:

```
<?Fm Condstart tag?>
```

while the one following the conditional text has the form:

```
<?Fm Condend tag?>
```

where *tag* is the condition tag. If multiple condition tags apply to the same content, each tag has separate processing instructions.

For example, if a list of recipes uses condition tags to identify recipes with particular ingredients, the list can be customized for people with various food allergies. Such a document might contain markup such as:

```
<?Fm Condstart fruit?>
<?Fm Condstart blueberry?>
<recipe>Blueberry Muffins</recipe>
<?Fm Condend blueberry?>
<?Fm Condstart apple?><?Fm Condstart orange?>
<recipe>Apple Orange Fruit Cup</recipe>
<?FM Condend orange?><?Fm Condend apple?>
<?Fm Condend fruit?>
```

On export to markup

By default, FrameMaker exports processing instructions that define all condition settings. When exporting a book, FrameMaker exports the condition settings for each book component after the start-tag for the book component.

FrameMaker exports all conditional text, with surrounding processing instructions, regardless of whether the conditional text is shown or hidden.

When hidden conditional text is exported, the result may not be a valid XML document. Suppose, for example, that a course catalog begins with a `Title` element, and that only one `Title` is permitted. Class offerings for different times of year might be edited in one conditional document. The author might edit the titles for different versions as conditional text within a single `Title` element or as different conditional `Title` elements. If the author takes the first approach, a valid XML document might include the following fragment:

```
<Title>
<?Fm Condstart Summer?>Summer<?Fm Condend Summer?>
<?Fm Condstart Winter?>Winter<?Fm Condend Winter?>
Course Catalog
</Title>
```

However, the author might choose to enter different complete `Title` elements, making each of them conditional. In this case, the exported XML document might include the following:

```
<?Fm Condstart Summer?>
<Title>Summer Course Catalog</Title>
<?Fm Condend Summer?>
<?Fm Condstart Winter?>
<Title>Winter Course Catalog</Title>
<?Fm Condend Winter?>
```

Since this fragment contains two `Title` elements where only one is permitted, the XML document is not valid. If you expect such usage in your environment, you might want to

permit multiple Title elements in your DTD. If you expect only one Title to be shown at a time, the EDD can still permit only a single Title. The use of slightly different models in FrameMaker and XML allows FrameMaker documents to be validated prior to publishing to confirm that the show/hide settings produce a document that conforms to the permitted structure. At the same time, it supports export of hidden conditional text to valid XML.

On import to FrameMaker

When FrameMaker opens an XML document containing processing instructions for conditional text, it sets condition settings according to the encountered <?Fm Condition?> processing instructions. If multiple processing instructions refer to the same condition tag, FrameMaker uses the first definition. If opening the XML document creates a FrameMaker book, FrameMaker uses the first processing instruction for each condition tag after the start-tag for each book component. Thus, a condition tag can have different condition settings in different book components.

If you import an XML document into an existing FrameMaker document, condition settings in the FrameMaker document have priority over any defined in the imported XML document. In particular, when creating an XML text inset, FrameMaker ignores any <?Fm Condition?> processing instructions for existing condition tags but defines new condition tags for other <?Fm Condition?> processing instructions.

FrameMaker applies the condition tags specified in <?Fm Condstart?> and <?Fm Condend?> to the indicated content.

Modifications to the default translation

When FrameMaker imports an XML document, it interprets all conditional text processing instructions it encounters. For export, you can control:

- Whether hidden conditional text is exported
- Whether processing instructions delimit conditional text, that is, whether <?Fm Condstart?> and <?Fm Condend?> processing instructions are written

You control these options in the application definition in structapps.fm or another application definition file rather than with read/write rules. Add a ConditionalText child to the XMLApplication element for the application you are using. Place a single child, OutputTextPI in this element. Then add one of the four children listed in the following table to the OutputTextPI element:

Child of OutputTextPi	FrameMaker outputs hidden conditional text	Processing instructions delimit conditional text
OutputAllTextWithPIs	yes	yes
OutputAllTextWithoutPIs	yes	no
OutputVisibleTextWithPIs	no	yes
OutputVisibleTextWithoutPIs	no	no

After you edit the application definition file, use File > Structure Tools > Read Application Definitions or restart FrameMaker to enable the option you select.

24 Processing Multiple Files as Books

FrameMaker provides a mechanism for grouping multiple FrameMaker documents into a single unit called a book. Each document remains a separate, complete FrameMaker document, but FrameMaker provides a set of facilities for working with those documents as a unit. For example, you can easily number pages consecutively throughout the book, generate a table of contents for the entire book, print the entire book with one command, or validate the element structure for the entire book.

Markup doesn't explicitly provide such a mechanism. However, markup documents are sometimes divided into external text entities so that a large document can be distributed over several files.

This use of text entities as include files is analogous to a FrameMaker book file, but markup allows more freedom in specifying the files that correspond to a book:

- A FrameMaker book can have only one level of documents. A markup document can have more than one level of nesting.
- A document in a FrameMaker book can either be unstructured or a single, complete element. An external text entity can include a partial element.

Because of this difference, if you start with a markup document that doesn't match the FrameMaker model for books, the entity structure will not correspond to the book structure. On the other hand, any valid FrameMaker book can be easily translated to markup with each book component exported as a text entity.

In this chapter

> **XML and SGML:** The XML specification indicates that closing delimiter string for a PI is ?>. Unless otherwise stated, the examples of PIs in this section use the XML syntax.

This chapter starts by describing the default translation for books. The chapter then describes your options for modifying the translation. In the outline below, click a topic to go to its page.

How FrameMaker translates books and book components by default:

- "On import to FrameMaker" on page 348
- "On export to markup" on page 350

Some ways you can change the default translation:

- "Using elements to identify book components on import" on page 351
- "Suppressing the creation of processing instructions for a book on export" on page 353

Default translation

Books in FrameMaker and their counterparts in markup don't have special structure associated with them in the EDD or DTD. For this reason, creating an EDD or a DTD doesn't add book-specific information. All translation is done during conversion of individual FrameMaker books or markup documents.

On import to FrameMaker

FrameMaker does not attempt to automatically subdivide a markup document into FrameMaker documents in a book. However, you can use processing instructions or read/write rules to signal that a file is a book file and to signal the start of each new document in the book.

> **XML and SGML:** The XML specification defines the PI closing delimiter as ?>, while in SGML the closing delimiter is >. Also, for XML the colon is omitted. For example, following are two processing equivalent instructions—in SGML, <?FM: book>; in XML, <?FM book?>.
>
> This section uses the XML specification to illustrate PI syntax.

To instruct FrameMaker to generate a book, you can place the following processing instruction before the start-tag of the document element of a markup document:

```
<?FM book?>
```

The book processing instruction may occur before, within, or after the DTD. It is an error if it occurs elsewhere, if it is preceded by a document processing instruction (described next), or if it occurs more than once.

Within the markup document, you indicate the start of a new document in a FrameMaker book by placing a processing instruction immediately before the start-tag of the element that is the highest-level element in the FrameMaker document. The processing instruction has the form:

```
<?FM document "fname"?>
```

where *fname* is the name of the new document file. If *fname* is a relative pathname, it is relative to the directory for the book file. Any spaces within *fname* must be escaped by a backslash. If *fname* contains the processing instruction close delimiter string (specified as > in the SGML reference concrete syntax, and as ?> in XML), the processing instruction must be entered through an entity. On export, FrameMaker generates *fname* from the name of the file it is processing.

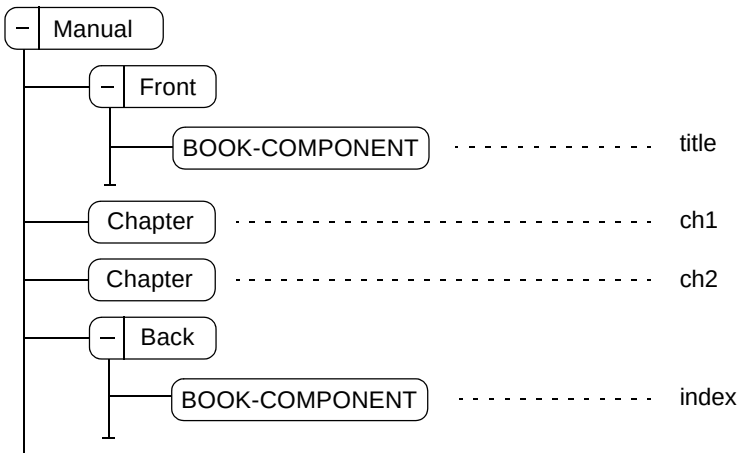
You cannot omit the end-tag for an element that immediately precedes a processing instruction, even if markup minimization is allowed (in SGML). If you did so, the parser

would place the omitted end-tag before the start-tag for the next element. Since this start-tag appears after the processing instruction, the end-tag generates an error in the log file.

For example, assume you have a book broken into four parts—the front matter, two chapters, and an index. In FrameMaker, each part is in a separate document. In SGML this situation might be represented as:

```
<!DOCTYPE manual SYSTEM "manual.dtd"
<?FM boo?>
<manual>
<?FM document "title"?>
<front>
. . .
</front>
<?FM document "ch1"?>
<chapter>
. . .
</chapter>
<?FM document "ch2"?>
<chapter>
. . .
</chapter>
<?FM document "index"?>
<back>
. . .
</back>
</manual>
```

In FrameMaker this SGML structure appears as:



On export to markup

FrameMaker treats documents in a FrameMaker book as external text entities. In addition to the file it creates for the entire book (the document entity), it creates a separate file for each structured document. It also creates separate files for unstructured documents as necessary and generates appropriate entity references for these additional files. FrameMaker also produces the processing instructions that allow it to recreate the original book structure.

In particular, FrameMaker names the entities corresponding to documents in a book `bkc1`, `bkc2`, and so on. If an entity name conflicts with the name of an existing entity, FrameMaker increments the counter and tries again. For example, if your DTD already defines an entity `bkc1`, FrameMaker tries instead to name the first book component entity `bkc2`.

Each book component entity declaration includes an external identifier containing a system identifier derived from the document name by dropping any extension and adding a new extension of the form `.e01`, `.e02`, `.e03`, and so forth. If there are more than 100 book components, the extension has the form `.001`, `.002`, `.003`, and so forth.

For example, assume you have the book file with the structure shown earlier. On export, FrameMaker generates this SGML document for that structure:

```
<!DOCTYPE manual . . . [
<!--Begin Document Specific Declarations-->
<!ENTITY bkc1 SYSTEM "title.e01">
<!ENTITY bkc2 SYSTEM "ch1.e02">
<!ENTITY bkc3 SYSTEM "ch2.e03">
<!ENTITY bkc4 SYSTEM "index.e04">
<!--End Document Specific Declarations-->
. . . other local entity declarations . . .]
]>
<?FM book?>
<manual>
<front>
&bkc1;
</front>
&bkc2;
&bkc3;
<back>
&bkc4;
</back>
</manual>
```

Each text entity starts with the appropriate document processing instruction. Thus, the entity `bkc2` has the form:

```
<?FM document ch1?>
<chapter>
. . .
</chapter>
```

When FrameMaker exports a book, it creates a new file for each structured document. Unstructured documents appear as text within an element.

In general, you can use a different EDD for each document in a FrameMaker book. However, if you export the book to markup, the result will be invalid unless the DTD provides a superset of the element declarations used in all the EDDs.

Modifications to the default translation

FrameMaker provides a small number of read/write rules specific to translating books. In addition, there are rules relevant to translating files within books. If you need to make other changes to the translation, you write a structure API client.

To specify the handling of unstructured documents, you specify the handling of the containing element.

- To discard a document, use the `drop` rule described in "drop" on page 373.
- To import or export a document as an empty element, use the `drop content` rule described in "drop content" on page 375.

For a summary of read/write rules relevant to translating books and book components, see "Books" on page 356.

Using elements to identify book components on import

The default method to specify how a markup document should be broken into components of a FrameMaker book is with the processing instructions described above. However, you may be able to use rules for this purpose instead.

If you can identify elements in your DTD that correspond to where you want file breaks in FrameMaker, you can use this rule:

```
reader generate book
   put element "gi" in file ["fname"];
```

For example, if you want instances of the `chapter` and `section` elements to indicate new files, you can use these rules:

```
reader generate book {
   put element "section" in file "Sect.fm";
   put element "chapter" in file;
}
```

This will cause occurrences of `section` elements to create files named `Sect1.fm`, `Sect2.fm`, and so on and occurrences of `chapter` elements to create files named `chapter1.doc`, `chapter2.doc`, and so on.

FrameMaker does not create a new file if the element occurs inside an element that already created a new file. For example, if you use the `section` element both to indicate a new file and to indicate sections within a file, you can still use the rules above. A `section` element inside a `chapter` element or another `section` element does not create an additional file.

With this form of the rule, FrameMaker always creates a book for these elements. If you want FrameMaker to create a book only when it is processing documents with particular document elements, use this form of the rule:

```
reader generate book for doctype "dt1" [. . . "dtN"]
   put element "gi" in file ["fname"];
```

For example, if you want FrameMaker to create a book only if the document element is `reference` or `manual`, you can use this rule:

```
reader generate book for doctype "reference", "manual" {
   put element "section" in file "Sect";
   put element "chapter" in file;
}
```

With this rule, if you import this SGML document:

```
<!DOCTYPE reference
. . .
<reference>
<chapter>
<section>Intro
. . .
</section>
. . .
</chapter>
</reference>
```

FrameMaker creates two files: a book file for the `reference` element, and a document file for the `chapter` element. Since the `section` element occurs inside the `chapter` element,

the software doesn't create a separate document file for it. On the other hand, if you import this SGML document:

```
<!DOCTYPE chapter
. . .
<chapter>
<section>Intro
. . .
</section>
. . .
</chapter>
```

FrameMaker creates one file containing the entire structure; it does not create a separate book file.

For information on these rules, see "reader" on page 453, "generate book" on page 403, and "output book processing instructions" on page 447.

Suppressing the creation of processing instructions for a book on export

By default, FrameMaker creates processing instructions for books and book components whenever you export a FrameMaker book to markup. This happens even if you use the `generate book` rule to cause FrameMaker not to need the processing instructions on import. To keep FrameMaker from writing any book or book component processing instructions, use the following rule:

```
writer do not output book processing instructions;
```

To confirm FrameMaker's default behavior of creating these processing instructions, use this rule:

```
writer output book processing instructions;
```

For information on these rules, see "writer" on page 467 and "output book processing instructions" on page 447.

25 Read/Write Rules Summary

This chapter lists the available read/write rules by category and briefly describes the purpose of each rule. The categories, which are arranged alphabetically, are as follows: all elements, attributes, books, entities, cross-references, footnotes, equations, markers, graphics, processing instructions, markup documents, tables, text, text insets, and variables.

All Elements

To	Use this rule	Page
Translate a markup element	element	376
Discard or unwrap a FrameMaker element on export	`fm element`	397
Translate a markup element to a FrameMaker element	`is fm element`	419
Translate a markup attribute within the context of a single markup element	attribute	367
Inform FrameMaker not to update a FrameMaker element's definition when updating an existing EDD	preserve fm element definition	448
Discard a FrameMaker or markup element	drop	373
Discard the content but not the structure of a FrameMaker or markup element	drop content	375
Discard the structure but not the content of a markup or FrameMaker element	unwrap	461

Attributes

To	Use this rule	Page
Translate a markup attribute	attribute	367
Discard a FrameMaker attribute	`fm attribute`	396
Translate a markup attribute to a FrameMaker attribute	`is fm attribute`	414
Translate a markup attribute within the context of a single markup element	element	376
Discard a markup or FrameMaker attribute	drop	373

To	Use this rule	Page
Translate a markup attribute to a particular FrameMaker property	`is fm property`	424
Translate a value for a markup attribute to a FrameMaker property value	`is fm property value`	426
Translate a value of a markup notation attribute or name token group to a value for a FrameMaker choice attribute	`is fm value`	439
Translate a markup attribute value to a FrameMaker property or a choice attribute value	value	464
Specify the value to use for a markup implied attribute when a document instance provides no value	`implied value is`	406

Books

To	Use this rule	Page
Specify whether to use elements or processing instructions to indicate book components when reading a markup document	`generate book`	403
Specify elements to use to indicate book components when reading a markup document	`put element` (described with `generate book`)	403
Specify the use of processing instructions to indicate book components when reading a markup document	`use processing instructions` (described with `generate book`)	403
Specify whether or not to write processing instructions that indicate book components in a markup document	output book processing instructions	447

Cross-references

To	Use this rule	Page
Translate markup elements to FrameMaker cross-reference elements	`is fm cross-reference element`	418
Translate FrameMaker cross-reference properties when no markup attribute exists	`fm property`	399
Translate FrameMaker cross-reference properties when no markup attribute exists	`value is` (described with `fm property`)	399

To	Use this rule	Page
Translate a markup attribute to a particular FrameMaker property	`is fm property`	424
Translate a value for a markup attribute to a FrameMaker property value	`is fm property value`	426
Translate a value of a markup notation attribute or name token group to a value for a FrameMaker choice attribute	`is fm value`	439
Translate a FrameMaker cross-reference element to text in markup	`fm element` unwrap	397, 461

Entities

To	Use this rule	Page
Translate a markup entity reference to an appropriate FrameMaker representation	entity	381
Determine the form of names of entities created for exported graphics	entity name is	383
Drop references to external data entities	`external data entity reference`	391
Translate an entity reference to a FrameMaker variable	`is fm variable`	441
Translate an entity reference to a single character	`is fm char`	416
Translate an entity reference to an element on a reference page	`is fm reference element`	430
Translate an SDATA entity reference to a text inset	`is fm text inset`	437
Determine the formatting of a text inset	reformat as plain text	454
	reformat using target document catalogs	455
	retain source document formatting	456
Discard external data entity references	drop	373

Equations

To	Use this rule	Page
Translate a markup element to a FrameMaker equation element	`is fm equation element`	420

To	Use this rule	Page
Specify export information for translating FrameMaker equations	equation	385
Specify the filename used for exporting an equation	`export to file`	389
Determine the form of names of entities created for exported equations	entity name is	383
Specify the data content notation for an exported equation	notation is	445
Determine whether FrameMaker uses the `dpi` attribute or the `impsize` attribute for equations and also the resolution used	specify size in	456
Translate FrameMaker cross-reference properties when no markup attribute exists	`fm property`	399
Translate FrameMaker cross-reference properties when no markup attribute exists	`value is` (described with `fm property`)	399
Translate FrameMaker equation properties to markup attributes	`is fm property`	424
Translate a value for a markup attribute to a FrameMaker property value	`is fm property value`	426
Translate a value of a markup notation attribute or name token group to a value for a FrameMaker choice attribute	`is fm value`	439
Translate a markup attribute value to a FrameMaker property or a choice attribute value	value	464

Footnotes

To	Use this rule	Page
Translate a markup element to a FrameMaker footnote element	`is fm footnote element`	421

Graphics

To	Use this rule	Page
Translate a markup element to a FrameMaker graphic element	`is fm graphic element`	422
Specify export information for translating FrameMaker graphics	anchored frame	365

To	Use this rule	Page
Specify export information for translating FrameMaker graphics that have a single inset	`facet`	394
Specify the filename used for exporting a graphic or a facet of a graphic	`export to file`	389
Force the software to export graphic files that were imported by reference	convert referenced graphics	372
Determine the form of names of entities created for exported graphics	entity name is	383
Specify the data content notation for an exported graphic	notation is	445
Determine whether FrameMaker uses the `dpi` attribute or the `impsize` attribute for imported graphics objects and also the resolution used	specify size in	456
Translate FrameMaker cross-reference properties when no markup attribute exists	`fm property`	399
Translate FrameMaker cross-reference properties when no markup attribute exists	`value is` (described with `fm property`)	399
Translate FrameMaker graphic properties to markup attributes	`is fm property`	424
Translate a value for a markup attribute to a FrameMaker property value	`is fm property value`	426
Translate a value of a markup notation attribute or name token group to a value for a FrameMaker choice attribute	`is fm value`	439
Translate a markup attribute value to a FrameMaker property or a choice attribute value	value	464

Markers

To	Use this rule	Page
Discard FrameMaker non-element markers or translate them to processing instructions	`fm marker`	398
Translate a markup element to a FrameMaker marker element	`is fm marker element`	423

To	Use this rule	Page
Determine whether marker text for marker elements becomes content or an attribute value in markup	marker text is	444
Drop references to external data entities	external data entity reference	391
Drop unrecognized processing instructions	processing instruction	451
Translate FrameMaker non-element markers to processing instructions	is processing instruction	441
Discard non-element markers	drop	373
Translate FrameMaker cross-reference properties when no markup attribute exists	fm property	399
Translate FrameMaker cross-reference properties when no markup attribute exists	value is (described with fm property)	399
Translate FrameMaker marker properties to markup attributes	is fm property	424
Translate a value for a markup attribute to a FrameMaker property value	is fm property value	426
Translate a value of a markup notation attribute or name token group to a value for a FrameMaker choice attribute	is fm value	439
Translate a markup attribute value to a FrameMaker property or a choice attribute value	value	464

Processing instructions

To	Use this rule	Page
Specify the treatment of unrecognized processing instructions	processing instruction	451
Specify the use of processing instructions to indicate book components when reading a markup document	use processing instructions (described with generate book)	403
Specify whether or not to write processing instructions that indicate book components in a markup document	output book processing instructions	447
Translate FrameMaker non-element markers to specific markup, or drop them	fm marker	398
Translate FrameMaker non-element markers to processing instructions	is processing instruction	441

To	Use this rule	Page
Discard processing instructions	drop	373

Markup documents

To	Use this rule	Page
Specify whether or not to use an external DTD subset to contain the DTD for a markup document created by FrameMaker	`include dtd`	408
Specify whether or not to include an SGML declaration in an SGML document created by FrameMaker	`include sgml declaration`	409
Specify the system and public identifiers for an external DTD subset	`external dtd`	392
Specify whether to create an entire markup document or just a markup document instance	`write structured document`	466
	`write structured document instance only`	466

Tables

To	Use this rule	Page
Translate a markup element to a FrameMaker table element	`is fm table element`	435
Translate a markup element to a FrameMaker element for a particular table part	`is fm table part element`	436
When creating a FrameMaker table, insert a table part even if that part is empty	`insert table part element`	410
Specify that a particular element always indicates a new table row	start new row	458
Indicate the start of a vertical straddle	start vertical straddle	460
Indicate the end of a vertical straddle	end vertical straddle	379
Specify the ruling style used for all tables	table ruling style is	461
Specify the resolution used for column widths with proportional widths	proportional width resolution is	452
Specify that the software write the width of table columns using proportional units	use proportional widths	463
Translate FrameMaker table properties to markup attributes	`is fm property`	424

To	Use this rule	Page
Translate a value for a markup attribute to a FrameMaker property value	`is fm property value`	426
Translate a value of a markup notation attribute or name token group to a value for a FrameMaker choice attribute	`is fm value`	439
Translate a attribute's name token value to a FrameMaker property or choice value	value	464

Text

To	Use this rule	Page
Translate an entity reference to a single character	`is fm char`	416
Determine the treatment of line breaks in reading and writing markup documents	line break	443
Define mappings between characters in the markup and FrameMaker character sets	character map	369

Text insets

To	Use this rule	Page
Translate an SDATA entity reference to a FrameMaker text inset	entity	381
	`is fm text inset`	437
Determine the formatting of a text inset	reformat as plain text	454
	reformat using target document catalogs	455
	retain source document formatting	456

Variables

To	Use this rule	Page
Translate a markup element to a FrameMaker system variable element	`is fm system variable element`	433
Translate an entity reference to a FrameMaker variable	`is fm variable`	441
Translate a markup entity reference to a FrameMaker variable	entity	381

To	Use this rule	Page
Determine treatment of FrameMaker non-element variables	`fm variable`	402
Translate a FrameMaker system variable element to text in markup	`fm element` unwrap	397, 461
Discard nonelement variables	drop	373

26 Read/Write Rules Reference

This chapter provides a reference to all read/write rules, listed in alphabetical order. The entry for each rule starts with a brief explanation of the purpose of the rule and how to use it. The rule's description may include the following sections:

Synopsis and contexts The rule's syntax and the context in which it can be used. If the rule occurs as a subrule of another rule, the more general rule is shown. If the rule can be used in multiple contexts, the synopsis shows each context. Each entry in this section shows a valid rule that has the current rule either at the highest level or as one of its subrules.

Rule synopses use the following conventions:

- Bold portions and nonitalicized portions of a rule are entered by you as shown.
- Italicized portions of a rule indicate the rule's arguments or possible subrules; you enter your values.
- Brackets [ ] indicate optional parts of a rule; the entire form within the brackets can be included or omitted.

Arguments The possible arguments to the rule. If an argument is optional, its default value is provided. Some rules have *subrule* as one of their arguments. In these cases, a list of possible subrules is provided. Some rule arguments allow variables. In these cases, a list of possible variables is provided.

Details Details on how to use the rule and on FrameMaker behavior when the rule is not supplied.

Examples Various examples of the rule.

See also Cross-references to other relevant information in the manual.

For information on how to create an s rules file and on the syntax of rules, see Chapter 14, "Read/Write Rules and Their Syntax."

anchored frame

Use the `anchored frame` rule only in an `element` rule for a graphic element, to provide information the software needs when writing a document that contains graphics to markup. For the circumstances when FrameMaker will write out the anchored frame contents as a graphic file, you use this rule to specify information such as graphic file format or notation name in subrules. Use this rule to specify information about the file FrameMaker creates to

contain the external data entity—for instances that are not anchored frames containing only a single imported graphic object.

Synopsis and contexts

```
element "gi" {
   is fm graphic element ["fmtag"];
   writer anchored frame subrule;
. . .}
```

Arguments

gi — A generic identifier.

fmtag — A FrameMaker element tag.

subrule — An `anchored frame` rule can have the following subrules:

`entity name is`, page 383, tells the software how to create the base name for the entity associated with this element type.

`export to file`, page 389, tells FrameMaker how to write the file name when it creates a new graphic file, and optionally the graphic format for the file.

`notation is`, page 445, specifies the data content notation of the entity file.

`specify size in`, page 456, specifies the units to use when writing the file.

Details

On export, if the anchored frame contains only a single imported graphic file, FrameMaker uses that graphic file for the resulting markup graphic element by default. If the anchored frame contains more than one graphic file, or has been modified using FrameMaker graphics tools, the software writes out a graphic file to be used. The default format for these graphic files is CGM. The export format can be changed with the `export to file` rule. For more information about translating anchored frame contents, see Chapter 19, "Translating Graphics and Equations."

Examples

- Assume you use the `Graphic` element for all graphic elements. If the graphic contains any single facet, assume the graphic was imported as an entity and you want the default behavior. However, if the author used FrameMaker graphic tools to create the objects in the graphic element, you want the file written in QuickDraw PICT format.

To accomplish all this, use this rule:

```
element "graphic" {
   is fm graphic element;
   writer anchored frame export to file "$(docname).pic"
      as "PICT";
}
```

Assume the FrameMaker document is named `mydoc.fm`. For the first graphic that is not a single facet, the software writes out a graphic file named `mydoc1.pic` in the PICT format.

If the export DTD declares an entity attribute to identify the graphic file with the `graphic` element, the software generates the following entity declaration:

```
<!ENTITY graphic1 SYSTEM "mydoc1.pic" NDATA PICT>
```

If the export DTD includes only a `file` attribute to associate the graphic file with the graphic element, the software uses this filename as its value.

See also

Related rules	“equation” on page 385 “facet” on page 394
Rules mentioned in synopses	“element” on page 376 “is fm equation element” on page 420 “is fm graphic element” on page 422 “writer” on page 467
General information on this topic	Chapter 19, “Translating Graphics and Equations”

attribute

Use the `attribute` rule to describe how to process a markup attribute. By default, a markup attribute translates to a FrameMaker attribute of the same name. Usually, this rule occurs as a subrule of the `element` rule, to describe treatment of the attribute *`attr`* within the element *`gi`*.

Synopsis and contexts

```
1. [structdv] attribute "attr" {. . .
      subrule;
   . . .}

2. element "gi" {. . .
      [structdv] attribute "attr" {. . .
         subrule;
   . . .} . . .}
```

Arguments

structdv — An optional markup declared value, specifying the type of the markup attribute. Legal values are: `cdata`, `name`, `names`, `nmtoken`, `nmtokens`, `number`, `numbers`, `nutoken`, `nutokens`, `entity`, `entities`, `notation`, `id`, `idref`, `idrefs`, and `group`.

attr — The name of a markup attribute.

gi — A generic identifier.

subrule — An `attribute` rule can have one or more of the following subrules:

`drop`, page 373, discards the attribute.

`implied value is`, page 406, specifies the value to use for an impliable attribute for which no value is given in a document instance.

`is fm attribute`, page 414, translates a markup attribute into a FrameMaker attribute.

`is fm property`, page 424, translates a markup attribute to a FrameMaker property such as the width of columns in a table. This subrule is applicable only to cross-reference, marker, graphic, equation, table, and table part elements.

`value`, page 464, translates one of the possible values of an SGML name token group or a notation attribute to a specific token of a FrameMaker choice attribute.

Details

- In some cases, the same attribute may occur in several markup elements and may require the same treatment for most of those occurrences. In these situations, you can use the `attribute` rule at the highest level to set the default treatment of the attribute. You can then override the default in individual `element` rules.
- The `drop`, `is fm attribute`, and `is fm property` subrules of the `attribute` rule are mutually exclusive. That is, if you use one of these rules, you cannot use either of the other rules.

Examples

- The following rule specifies that the `sec` attribute of the markup `list` element is in a name token group and corresponds to the attribute `Security` on the corresponding FrameMaker element:

```
element "list"
   group attribute "sec"
      is fm attribute "Security";
```

- Assume you have several elements that represent graphic objects. Each of them has an attribute `w`, representing the width of the object. Use this rule to make the width be 3 inches unless otherwise specified for a particular element:

```
attribute "w" {
   is fm property width;
   implied value is "3in";
}
```

- Assume you have an element `team` with an attribute `color`. The possible values for `color` are `r`, `b`, and `g`. To change the names of these values in the corresponding FrameMaker choice attribute, use this rule:

```
element "team" {
 attribute "color" {
      value "r" is fm value "Red";
      value "b" is fm value "Blue";
      value "g" is fm value "Green";
}}
```

See also

Related rules	"fm attribute" on page 396 "is fm attribute" on page 414
Rules mentioned in synopses	"element" on page 376
General information on this topic	Chapter 16, "Translating Elements and Their Attributes"

character map

Use the `character map` rule to define mappings between characters in the markup and FrameMaker character sets. Many characters can be expressed using a string; others require using the appropriate integer character code.

> **XML:** This read/write rule is primarily for SGML, only. For XML it is best to use UNICODE characters—by default FrameMaker assumes UTF-8 encoding for XML import and export. The use of UNICODE makes this rule unnecessary. If you want to use ISOLatin encoding with an XML document, then you may need to use this rule to map characters.

Synopsis and contexts

1. **character map** is $cmap_1$ [, . . ., $cmap_n$];
2. reader **character map** is $cmap_1$ [, . . ., $cmap_n$];
3. writer **character map** is $cmap_1$ [, . . ., $cmap_n$];

Arguments

cmap$_i$

A mapping between the character set used in the markup document and the FrameMaker character set. Each *cmap*$_i$ has one of the following forms:

```
sgmlch = fmch;
sgmlch = trap;
trap = fmch;
```

sgmlch is either a 1-character string or a character code representing a character in the markup character set. *sgmlch* can be a single character only if that character has the same character code in both the FrameMaker and markup character sets. Otherwise, you must use the integer character code.

fmch is either a 1-character string or a character code representing a character in the FrameMaker character set.

For information on how to represent character codes and special characters in strings, see "Strings and constants" on page 212.

Details

- Some characters might be defined in only one of the two character sets. The keyword `trap` is provided for this situation. By default, FrameMaker discards trapped characters.
- The character map need not be a one-to-one mapping. If a character in the input document is mapped to multiple characters in the output character set, FrameMaker uses the output character from the *last* mapping to appear in the `character map` rule.
- If you use the `character map` rule at the highest level, do not also use it inside either a `reader` rule or a `writer` rule. If you use this rule inside a `reader` rule or a `writer` rule and also use it at the highest level, FrameMaker ignores the highest-level `character map` rule. You can only have one occurrence of this rule at the highest level.

  Similarly, the `character map` rule can appear in one `reader` rule and one `writer` rule at most. The software ignores any subsequent uses of the `character map` rule.
- If you use the `character map` rule at the highest level, its behavior is bidirectional. For example, you could have this rule:

  ```
  character map is 0x20 = 0x12;
  ```

  This rule specifies that the ISO Latin-1 space character (character code `0x20`) maps to the FrameMaker thin space character (character code `0x12`). With this rule, FrameMaker translates a thin space to a standard space when it writes a markup document. However, this rule translates *all* spaces in a markup document to thin spaces in a corresponding FrameMaker document. This is unlikely to be the desired behavior. For this reason, instead you should use this rule:

  ```
  reader character map is 0x20 = 0x12;
  ```

- By default, FrameMaker assumes that the character set your SGML documents use is ISO Latin-1. It provides a default mapping between those character sets. For details, see Appendix E, "Character Set Mapping." For information on other character sets you can use, see Appendix F, "ISO Public Entities."
- By default, on export FrameMaker produces a character in the SGML document for most printing characters in the corresponding FrameMaker document. FrameMaker documents occasionally include unusual characters that serve no purpose outside FrameMaker. For example, the codes `0x01` and `0x03` are nonprinting characters that represent information about the insertion point movement. On export FrameMaker traps such characters, so that they don't appear in an exported SGML document.

  Similarly, on import FrameMaker produces a character in the FrameMaker document for most printing characters. It traps all control characters other than a tab or newline character.
- FrameMaker has an 8-bit character set. The SGML declaration can specify any character set that the SGML parser can handle. Because part of the character set description in the SGML declaration is not human-readable and may not be interpretable automatically, any differences between the native FrameMaker character set and the character set in the SGML document must be specified with the `character map` rule.
- By default, FrameMaker discards trapped characters. You can provide an structure API client to change the processing of trapped characters. For information on creating an structure API client, see the *Structure Import/Export API Programmer's Guide*.

Examples

- Both the FrameMaker and default SGML character sets have a character code for the character ó (lowercase o with an acute accent). In FrameMaker, the character code is `0x97`; in the default SGML character set, the character code is `0xF3`. If you want to trap the SGML character that looks like ó, you might try using this rule:

  ```
  character map is "ó" = trap;
  ```

  Because you enter your read/write rules in a FrameMaker document, however, FrameMaker interprets that rule as:

  ```
  character map is 0x97 = trap;
  ```

  which is not the behavior you want. Instead, you should use this rule:

  ```
  character map is 0xF3 = trap;
  ```
- By default, FrameMaker maps the SGML broken bar character to the FrameMaker solid bar character |. The rule for doing so could be written in the following equivalent ways:

  ```
  character map is 0xA6 = "|";
  character map is 0xA6 = 0x7C;
  character map is "\xA6" = "\x7C";
  ```
- To trap the SGML broken bar character, use this rule:

  ```
  character map is 0xA6 = trap;
  ```

See also

- For information on the FrameMaker character set, see the FrameMaker user's manual.
- For details of the default mapping between the FrameMaker and ISO Latin-1 character sets, see Appendix E, "Character Set Mapping."

convert referenced graphics

Use the `convert referenced graphics` rule to force the software to write out a graphic file when exporting a graphic element that uses a referenced graphic. By default, FrameMaker doesn't write out graphic files in this case. It is usually more advantageous to simply reference the same graphic file in both the markup and the FrameMaker document. However, you can use this rule to convert all such graphic files to a specific format.

Synopsis and contexts

```
element "gi" { . . .
   writer facet "facetname" convert referenced graphics;
   . . . }
```

Arguments

There are no arguments for this rule

Details

- This rule must be a subrule of a facet rule for a graphic element.
- By default, if a graphic or equation element is imported by reference, the software does not create a new graphic file for the element when exporting a FrameMaker document. You can change that behavior using this rule.

Examples

- Assume you want to convert imported graphic files in `graphic` elements which have not been edited in the FrameMaker document, to the PICT format. With the following example, the software would create PICT files for each of these graphic elements:

```
element "graphic" {
   is fm graphic element;

   writer {
   facet default {
      convert referenced graphics;
      export to file "$(entity).pic" as "PICT";
}}
```

- For graphic elements with a single TIFF facet, the following example converts the graphic files in the graphic element to PICT:

```
element "graphic" {
   is fm graphic element;
   writer facet "TIFF"{
      convert referenced graphics;
      export to file "$(entity).pic" as
         "PICT";
}}
```

See also

Related rules
"facet" on page 394
"export to file" on page 389
"writer" on page 467

General information on this topic
"Translating Graphics and Equations" on page 297

do not include dtd

See "include dtd" on page 408.

do not include sgml declaration

See "include sgml declaration" on page 409.

do not output book processing instructions

See "output book processing instructions" on page 447.

drop

Use the `drop` rule to indicate information that you want discarded. Examples of information you might discard include a markup element or attribute that has no counterpart in FrameMaker, or a FrameMaker non-element marker that has no counterpart in markup.

Synopsis and contexts

```
1. attribute "attr" drop;

2. element "gi" drop;

3. element "gi" {. . .
      attribute "attr" drop;
   . . .}
```

4. external data entity reference **drop**;

5. fm attribute "*attr*" **drop**;

6. fm element "*fmtag*" **drop**;

7. fm marker $type_1$ [, . . ., $type_n$] **drop**;

8. fm variable **drop**;

9. processing instruction **drop**;

Arguments

attr	The name of a markup or FrameMaker attribute. Note that `fm attribute` names are case-sensitive and should appear as in the EDD. The case of DTD attribute names depends on the setting of NAMECASE in the SGML.dcl file—For XML attribute names are case sensitive.
gi	A generic identifier.
fmtag	A FrameMaker element tag. These names are case-sensitive and should appear in the rule the same as in the EDD.
$type_i$	A FrameMaker marker type, such as Index or Type 22.

Details

- When FrameMaker encounters something to be discarded, it makes no attempt to insert the corresponding information into the document it is creating. In the case of a dropped element, it also discards all descendant elements.
- When creating an EDD from a DTD or a DTD from an EDD, FrameMaker does not generate an element definition corresponding to a dropped element. It also removes any references to the specified element in content rules for other elements unless you've specified a `preserve fm element definition` rule for those elements.
- You can write a structure API client to process dropped information. Your client must also handle retrieving discarded information if it is needed when the document is written back to its original format.
- If you use the `drop` rule in some rule, you can use no other subrules of the same rule. For example, you cannot specify that FrameMaker both drop an attribute and translate it to a FrameMaker property with the `is fm property` rule.

Examples

- A markup element used instead of a processing instruction to indicate that a page or line break is desired may be discarded when the markup document is read. Text formatting

rules in the EDD can be used to indicate a page break in FrameMaker; there is no need to mark the break with an element. To drop the markup element `break`, use this rule:

```
element "break" drop;
```

- By default, FrameMaker stores processing instructions that it does not recognize in non-element markers. In this way, even though FrameMaker does not perform special processing on the processing instruction, when you save the FrameMaker document back to markup, the software writes out the processing instruction so that a different application can use it. If you don't need to write out the processing instructions, you could use this rule:

```
processing instruction drop;
```

See also

Related rules	"drop content" on page 375 "unwrap" on page 461 "preserve fm element definition" on page 448
Rules mentioned in synopses	"attribute" on page 367 "element" on page 376 "external data entity reference" on page 391 "fm attribute" on page 396 "fm element" on page 397 "fm marker" on page 398 "fm variable" on page 402 "processing instruction" on page 451
General information on this topic	Chapter 16, "Translating Elements and Their Attributes"

drop content

Use the `drop content` rule to either create a FrameMaker empty element or a markup element with no content from occurrences of *`gi`*.

Synopsis and contexts

```
1. element "gi" {
       is fm element "fmtag";
       reader drop content;
   }

2. element "gi" {
       is fm element "fm tag";
       writer drop content;
   }
```

Arguments

gi	A generic identifier.

Details

- You can use this rule when you have an element whose content is created in a system-specific way. Because you plan to rely on some system to create the content, the existing content at the time you import or export a document may not be relevant. For example, you may have a markup element intended to contain a chapter number. In FrameMaker, you use FrameMaker's formatting capabilities to have the system maintain the value. When reading in the markup document, you can drop the current content of the number element.
- Use `drop content` inside a `reader` rule when you translate markup documents to FrameMaker documents. Use it inside a `writer` rule when you translate FrameMaker documents to markup.

Examples

- Assume your DTD has a `toc` element that represents the table of contents for a markup document. Because FrameMaker can automatically generate a table of contents, this markup element can have its contents dropped upon import.

```
element "toc" reader drop content;
```

- Assume the `total` element's content is computed by a structure API client. Outside the FrameMaker environment you will use a different program to perform the computation. Consequently, you do not want the value that is current when the document is exported. To discard the current value, use this rule:

```
element "total" writer drop content;
```

See also

Related rules	"drop" on page 373 "unwrap" on page 461
Rules mentioned in synopses	"element" on page 376 "reader" on page 453 "writer" on page 467
General information on this topic	Chapter 16, "Translating Elements and Their Attributes" *Structure Import/Export API Programmer's Guide*

element

You use the `element` rule as the primary rule for translating between a markup element and its corresponding FrameMaker representation.

Synopsis and contexts

```
element "gi" {. . .
   subrule;
. . .}
```

Arguments

gi — A generic identifier.

subrule — The subrules of `element` indicate the treatment of the markup element:

`attribute`, page 367, specifies what to do with a markup element's attributes.

`drop`, page 373, discards the element.

`fm attribute`, page 396, specifies what to do with attributes present in the FrameMaker representation of the element but not in the markup representation of it.

`fm property`, page 399, specifies what to do with FrameMaker properties associated with the element. This subrule applies only to elements that correspond to graphic, equation, table, table part, cross-reference, or marker elements.

`is fm colspec`, page HIDDEN, specifies that the element represents a CALS table colspec. This subrule applies only to CALS tables.

`is fm cross-reference element`, page 418, specifies that the element corresponds to a FrameMaker cross-reference element.

`is fm element`, page 419, translates the element to a particular FrameMaker element. You use this subrule to rename the element.

`is fm equation element`, page 420, specifies that the element corresponds to a FrameMaker equation element.

`is fm footnote element`, page 421, specifies that the element corresponds to a FrameMaker footnote element.

`is fm graphic element`, page 422, specifies that the element corresponds to a FrameMaker graphic element.

`is fm marker element`, page 423, specifies that the element corresponds to a FrameMaker marker element.

`is fm spanspec`, page HIDDEN, specifies that the element represents a CALS table spanspec. This subrule applies only to CALS tables.

`is fm system variable element`, page 433, specifies that the element corresponds to a FrameMaker system variable element.

`is fm table element`, page 435, specifies that the element corresponds to a FrameMaker table element.

`is fm table` *`part`* `element`, page 436, specifies that the element corresponds to a FrameMaker element for a particular table part, such as a table title or cell.

`marker text is`, page 444, specifies whether the text of a FrameMaker marker element should be element content or an attribute value in markup. This subrule applies only to marker elements.

`reader drop content`, page 375, specifies that the content but not the structure of an element should be discarded on import of a markup document.

`reader end vertical straddle`, page 379, indicates that the associated table cell or row element terminates a vertical table straddle. This subrule applies only to table cell or row elements.

`reader insert table` *`part`* `element`, page 410, indicates that the software should insert the specified table part (title, heading or footing), even if the markup element structure does not contain the corresponding element. This subrule applies only to table elements.

`reader line break`, page 443, determines whether to interpret line breaks in text segments in elements in the markup document as forced returns or spaces within the elements.

`reader start new row`, page 458, indicates that the occurrence of the associated table cell element always starts a new row in the table. This subrule applies only to table cell elements.

`reader start vertical straddle`, page 460, indicates that the associated table cell element starts a vertical table straddle. This subrule applies only to table cell elements.

`unwrap`, page 461, indicates that the content of the element, but not the element itself, should be included in the translated document.

`writer anchored frame`, page 365, tells FrameMaker what to do with graphic elements other than those with a single non-internal FrameMaker facet. This subrule applies only to graphic elements.

`writer drop content`, page 375, specifies that the content but not the structure of an element should be discarded on export of a FrameMaker document.

writer equation, page 385, tells FrameMaker what to do with equation elements. This subrule applies only to equation elements.

writer facet, page 394, tells FrameMaker what to do with a graphic element that has a single non-internal FrameMaker facet. This subrule applies only to graphic elements.

writer line break, page 443, limits the length of lines the software generates in the markup document.

writer notation, page 445, specifies a notation name when the element is a graphic or equation.

writer specify size in, page 456, specifies the units of measure for the size of a graphic or equation element.

Details

- If you use either the drop or unwrap subrule of an element rule, that subrule must be the element's only subrule. For example, you cannot both unwrap a markup element and translate it to a FrameMaker element.

Examples

- To translate the markup element p to the FrameMaker element Paragraph, use this rule:

```
element "p" is fm element "Paragraph";
```

- To translate the markup element tab2 to a FrameMaker table element Two Table with two columns, use this rule:

```
element "tab2" {
   is fm table element "Two Table";
   fm property columns value is "2";
}
```

See also

Related rules	"fm element" on page 397
General information on this topic	Chapter 16, "Translating Elements and Their Attributes"

end vertical straddle

Use the end vertical straddle rule inside the element rule for a table row or table cell to specify that the row (or the row containing the cell) indicates the end of a vertical straddle started by some earlier table cell element. The straddle can end either before the current row or at the current row.

Synopsis and contexts

```
element "gi" {
   is fm table row_or_cell element ["fmtag"];
   reader end vertical straddle "name1" [, . . . "namen"]
      [before this row];
. . .}
```

Arguments

gi	A generic identifier.
row_or_cell	One of the keywords: `row` or `cell`.
fmtag	A FrameMaker element tag.
$name_i$	A name associated with a table straddle. Each $name_i$ must occur in a corresponding `start vertical straddle` rule.

Details

- Your DTD may contain elements that you want to format as tables in FrameMaker even though the element hierarchy does not match that required by FrameMaker for tables. In such a situation, the nature of the element hierarchy may indicate where vertical straddles begin and end. The `end vertical straddle` rule allows you to specify such elements.
- Use this rule in conjunction with the `start vertical straddle` rule. That rule specifies a table cell that indicates the first cell in a vertical straddle. In the `start vertical straddle` rule, give a name to the particular straddle started by that element. In the `end vertical straddle` rule, you must specify by name which vertical straddles started by earlier cells are ended by the occurrence of *gi*.
- If you use this rule for a table cell element, you can end only one vertical straddle. If you use it for a table row element, you can end more than one vertical straddle.
- If you use this element without the `before this row` keyword phrase, the cell or row (*gi*) specified in the rule becomes part of the straddle. If you do include that keyword phrase, then the straddle ends in the row above the one specified.

Examples

- For an example of the use of this rule, see “Creating vertical straddles” on page 290.

See also

Related rules	“start vertical straddle” on page 460
General information on this topic	Chapter 18, “Translating Tables”

entity

You use the `entity` rule to translate an entity to an appropriate FrameMaker representation. With this rule, you can translate an entity to a particular character or set of characters, a reference element, a text inset, or a FrameMaker variable. If you choose to translate the entity to a text inset, you can also specify how to format that text inset in the resulting document.

Synopsis and contexts

```
1. entity "ename" {
       type_rule;
       [format_rule;]
   . . .}

2. reader entity "ename" {
       type_rule;
       [format_rule;]
   . . .}
```

Arguments

ename — An entity name.

type_rule — One of the following:

`is fm char`, page 416, translates the entity to a particular character in FrameMaker.

`is fm reference element`, page 430, translates the entity to an element whose content resides on a reference page in the FrameMaker document.

`is fm text inset`, page 437, translates the entity to a FrameMaker text inset.

`is fm variable`, page 441, translates the entity to a FrameMaker non-element variable.

format_rule — This subrule can be specified only if *type_rule* is `is fm text inset`. One of the following:

`reformat as plain text`, page 454, specifies that the software remove the internal structure and formatting from the text of the text inset and apply the formatting used at the insertion point.

`reformat using target document catalogs`, page 455, specifies that the software retain the text inset's internal structure and apply the containing document's formats and element format rules to the text. This rule is applied as if the following three options were checked when a file is imported through the File>ImportFile menu: 1.

Reformat Using Target Document's catalog; 2. While importing Remove: Manual Page Breaks; and 3. While Importing Remove: Other Format Overrides. For more information, see the section "Importing text" in Chapter 5 of the FrameMaker User Guide.

`retain source document formatting`, page 456, specifies that the software remove the internal structure of the text inset and retain the formatting of the text inset as it appeared in the source document.

Details

- If you use the `entity` rule at the highest level, then it applies both on import and export. If you use it inside a `reader` rule, then FrameMaker translates the entity as specified when importing a markup document, but does not create an entity reference on export.
- While you can use this rule to translate any entity type to a text inset, we recommend you convert only SDATA entities to text insets. Note that the source file for such a text inset must be a format FrameMaker can automatically filter. Also, such a text inset cannot use a markup document as the source file.
- For XML and SGML, FrameMaker imports external text entities as text insets by default. The source files for these insets can be markup or text files. The software stores entity information on the Entity Declarations reference page so it can export the text inset as an external text entity.
- For XML, SDATA and CDATA entities are not allowed.

Examples

- To translate the text entity mn to the FrameMaker variable `Manual Name`, use this rule:

  ```
  entity "mn" is fm variable "Manual Name";
  ```

  Suppose the text entity mn is declared as `<!ENTITY mn "Developer's Guide">`, and the template for the application doesn't contain a variable named `Manual Name`. Then the software will create a FrameMaker variable named `Manual Name` defined as `Developer's Guide` and replace the reference in the text with the variable text `Developer's Guide`.

  However, if a FrameMaker variable named `Manual Name`, defined for example as `My Favorite Manual`, currently exists in the template for the application, when importing SGML, the software will not create a new variable nor modify the existing one. It will replace the reference in the text with the variable text `My Favorite Manual`. When importing XML, it does modify the variable definition.

- To have FrameMaker create a text inset for the `legalese` entity using the text in the file `legal.fm` and to have the software format that text inset as it appears in `legal.doc`, use this rule:

```
entity "legalese" {
   is fm text inset "legal.fm";
   retain source document formatting;
}
```

See also

General information on this topic	Chapter 17, "Translating Entities and Processing Instructions" Chapter 21, "Translating Variables and System Variable Elements"

entity name is

Use the `entity name is` rule only in an `element` rule for a graphic or equation element to provide information the software needs when writing a document containing graphics or equations to markup. The `entity name is` rule determines the name FrameMaker gives an entity reference it generates for the graphic or equation.

Synopsis and contexts

```
element "gi" {
   is fm graphic_or_eqn element ["fmtag"];
   writer type ["facetname"] entity name is "ename";
. . .}}
```

Arguments

gi	A generic identifier.
graphic_or_eqn	One of the keywords: `graphic` or `equation`.
type	One of the rules `anchored frame`, `facet`, or `equation`. If `facet`, you must also supply the *facetname* argument. If *type* is `equation`, the rule applies to equation elements. If *type* is `facet`, the rule applies to a graphic element that contains only a single facet with the name specified by *facetname*. This occurs when the graphic element is an anchored frame containing only a single imported graphic object whose original file was in the *facetname* graphic format. You can use this rule with *type* set to `facet` multiple times if you want the software to treat several file formats differently. If *type* is `anchored frame`, the rule applies to a graphic element under all other circumstances.
facetname	A facet name. You supply this argument if and only if *type* is `facet`.

ename A string representing the base name for an entity name.

Details

- By default, when FrameMaker exports an external data entity for a graphic or equation, it uses the entity name that is stored with the graphic inset. If there is no such entity name, the software generates a name for the entity based on the element name. You use the `entity name is` rule to change this behavior.

  The entity name you specify is a base name FrameMaker uses to generate a unique entity name. When it needs to create a new entity name, the software adds an integer to the name specified by *ename* to create a unique name.

Examples

- Assume you have a markup element `graphic` that corresponds to graphic elements in FrameMaker. Suppose further that some of the graphic elements in FrameMaker contain imported-by-copy graphics, or contain modifications to a graphic inset using FrameMaker graphic tools, or contain just graphic objects drawn using FrameMaker graphic tools. On export, the software must create new graphic files for these elements and declare entities for them. By default, FrameMaker would declare entities for these graphic elements based on the element name "graphic," for example, `graphic1`, `graphic2`, and so on. To specify that the names of the entities associated with such successive graphic elements have the form `car1`, `car2`, and so on, use this rule:

```
element "graphic" {
   is fm graphic element;
   writer anchored frame entity name is "car";
}
```

- Assume with a single facet graphics in the `car` element sometimes use the IGES file format and sometimes use the TIFF file format. Also assume that the DTD for the application does not currently contain entity declarations for the imported-by-reference graphic files. By default, the software would declare entities for all such graphics based on the element name "car," for example, `car1`, `car2`, and so on. If you want to name the entities for the IGES graphics `icar` and the entities for the TIFF graphics `tcar`, then use this rule:

```
element "car" {
   is fm graphic element;
   writer facet "IGES" entity name is "icar";
   writer facet "TIFF" entity name is "tcar";
}
```

See also

Related rules
“export to file” on page 389
“notation is” on page 445
“specify size in” on page 456

Rules mentioned in synopses	“element” on page 376 “is fm graphic element” on page 422 “is fm equation element” on page 420 “anchored frame” on page 365 “equation” on page 385 “facet” on page 394 “writer” on page 467
General information on this topic	Chapter 19, “Translating Graphics and Equations”

equation

Use the `equation` rule only in an `element` rule for an equation element, to provide information the software needs when writing to markup a document containing equations. FrameMaker creates graphic files to represent equations. Use this rule to specify information about the files FrameMaker creates for instances of the equation element. By default, the software creates a file in CGM format for each equation, and the filename is based on the element name. Also, by default, if the equation element is associated with an external data entity, then the entity name is based on the element name.

Synopsis and contexts

```
element "gi" {
   is fm equation element ["fmtag"];
   writer equation subrule;
. . .}
```

Arguments

gi	A generic identifier.
fmtag	A FrameMaker element tag.
subrule	An `equation` rule can have the following subrules: `entity name is`, page 383, tells the software how to create the base name for the entity associated with this element type. `export to file`, page 389, tells the software to write a new file for the associated external data entity. `notation is`, page 445, specifies the data content notation of the entity file. `specify size in`, page 456, specifies the units to use when writing the file.

Examples

- Assume you have an element named `math` with an attribute of type `Entity` that is mapped to the `fm property` entity for this element. If you want to create TIFF files for the equations in a document named `mytest.doc`, you might use this rule:

```
element "math" {
   is fm equation element;
   writer equation export to file "$(docname).eqn" as "TIFF";
}
```

  The software will create graphic files for each equation in `mytest.doc` named `mytest1`, `mytest2`,...and will declare entities named `math1`, `math2`, ...for each graphic.

See also

Related rules	“anchored frame” on page 365 “facet” on page 394 “is fm graphic element” on page 422
Rules mentioned in synopses	“element” on page 376 “is fm equation element” on page 420 “writer” on page 467
General information on this topic	Chapter 19, “Translating Graphics and Equations”

export dpi is

You use the `export dpi` rule only in an `element` rule for a graphic or equation element, to provide information the software needs when writing a document containing graphics or equations to markup. The `export dpi` rule tells FrameMaker the dpi setting to use for an exported graphic file.

Synopsis and contexts

```
element "gi" {
   is fm graphic_or_eqn element ["fmtag"];
   writer type ["facetname"]
      export dpi is number;
      . . .
. . .}
```

Arguments

gi	A generic identifier.
graphic_or_eqn	One of the keywords: `graphic` or `equation`.
type	One of the rules `anchored frame`, `facet`, or `equation`. If `facet`, you must also supply the *facetname* argument. If *type* is `equation`, the rule applies to equation elements. If *type* is `facet`, the rule applies to a graphic element that contains only a single facet with the name specified by *facetname*. This occurs when the graphic element is an anchored frame containing only a single imported graphic object whose original file was in the *facetname* graphic format. You can use this rule with *type* set to `facet` multiple times if you want the software to treat several file formats differently. If *type* is `anchored frame`, the rule applies to a graphic element under all other circumstances.
facetname	A facet name. You supply this argument if and only if *type* is `facet`.
number	The value for your dpi setting.

Details

- In the absence of this rule, FrameMaker uses the dpi setting associated with the graphic file. If there is no setting associated with the graphic, the software assumes a value of 300.
- For Windows versions of FrameMaker, if the initialization file for a graphics filter specifies a dpi setting that setting overrides this rule whenever that filter is used to export a graphic file.

Examples

- Assume you export the FrameMaker file `Math.doc` and have the following rule:

```
element "eqn" {
   is fm equation element "Equation";
   writer equation
      export dpi is 72;
}
```

  When FrameMaker finds an instance of the `Equation` element, it exports equations as graphic files at 72 dpi.

- Assume you have the rule:

```
element "imp" {
   is fm graphic element;
   writer facet “TIFF”{
      convert referenced graphics;
      export dpi is 1200;
      export to file "$(entity).tif";
}}
```

  This rule tells FrameMaker for every graphic element with a single TIFF facet, it should write a new graphic file with a dpi of 1200, using the entity name as part of the graphic file’s filename.

See also

Related rules	“convert referenced graphics” on page 372 “entity name is” on page 383 “notation is” on page 445 “specify size in” on page 456
Rules mentioned in synopses	“element” on page 376 “is fm graphic element” on page 422 “is fm equation element” on page 420 “anchored frame” on page 365 “equation” on page 385 “facet” on page 394 “writer” on page 467
General information on this topic	Chapter 19, “Translating Graphics and Equations”

export to file

You use the `export to file` rule only in an `element` rule for a graphic or equation element, to provide information the software needs when writing a document containing graphics or equations to markup. The `export to file` rule tells FrameMaker how to write the file name when it creates a new graphic file, and optionally the graphic format for the file.

Synopsis and contexts

```
element "gi" {
   is fm graphic_or_eqn element ["fmtag"];
   writer type ["facetname"]
      export to file "fname" [as "format"];
. . .}
```

Arguments

gi	A generic identifier.
graphic_or_eqn	One of the keywords: `graphic` or `equation`.
type	One of the rules `anchored frame`, `facet`, or `equation`. If `facet`, you must also supply the *facetname* argument. If *type* is `equation`, the rule applies to equation elements. If *type* is `facet`, the rule applies to a graphic element that contains only a single facet with the name specified by *facetname*. This occurs when the graphic element is an anchored frame containing only a single imported graphic object whose original file was in the *facetname* graphic format. In this case, the rule is only executed if the “converted referenced graphics” rule is also used. Otherwise, it is ignored. You can use this rule with *type* set to `facet` multiple times if you want the software to treat several file formats differently. If *type* is `anchored frame`, the rule applies to a graphic element under all other circumstances and does not use the “convert referenced graphics” rule.
facetname	A facet name. You supply this argument if and only if *type* is `facet`. The string for the facetname must exactly match the string for the facetname in the FrameMaker document. To determine a graphic file’s facetname, select the graphic, click Graphics>ObjectProperties, and observe the facetname in the dialog box.
fname	A base filename which can be either absolute or relative to the output directory. Note: If path information is included in fname, the Windows platform requires double backslashes to translate correctly as path

backslashes. The *fname* argument can contain the variables $(docname) and $(entity), described below.

format — A file data content format code, such as TIFF or PICT. See Chapter 19, "Translating Graphics and Equations," for a complete list of graphic format codes. format must be one of these code names.

Details

- By default, if a graphic element has a single facet (other than a FrameMaker internal facet) that was imported by reference, FrameMaker does not create a new graphic file. On export, the original file will be associated with a markup graphic element via the file attribute, or via the entity attribute plus a corresponding entity declaration. You can use the convert referenced graphics rule to force FrameMaker to export such graphic files.
- If your rules specify the software will write a graphic file, if a graphic element has a single facet (other than a FrameMaker internal facet), FrameMaker writes the graphic file in that format by default. It writes the graphic file for equation elements and all other graphic elements in CGM format by default.

  If you supply a *format* argument, you must first make sure that the format is one known to FrameMaker. For information on which graphic export filters the software provides and on how to add new ones, see Chapter 19, "Translating Graphics and Equations".
- The *fname* argument can use these variables:

Variable	Meaning
$(entity)	The value of the corresponding markup element's entity attribute. If the source of the graphic inset wasn't originally an entity, this variable defaults to a unique name based on the name of the element. You can change this name using the "entity name is" rule.
$(docname)	The name of the FrameMaker file, excluding any extension or directory information.

- The *fname* argument is used as a template for the actual filename FrameMaker generates for a particular graphic or equation element. FrameMaker takes the filename specified with the *fname* argument and may append an integer to the filename to ensure uniqueness of the filename. For an example of this behavior, see the first example below.

Examples

- Assume you export the FrameMaker file Math.fm and have the following rule:

```
element "eqn" {
   is fm equation element "Equation";
   writer equation
      export to file "$(docname).eqn" as "PICT";
}
```

When FrameMaker finds an instance of the `Equation` element, it generates filenames of the form `Math`*`N`*`.eqn` until it finds a name that doesn't collide with an already existing file. For example, if you already have files in the specified directory named `Math1.eqn` and `Math2.eqn`, the software writes the first equation to a file named `Math3.eqn`. FrameMaker writes the equation file in PICT format, instead of the default CGM format.

- Assume you have the rule:

```
element "imp" {
   is fm graphic element;
   writer facet "TIFF" {
   convert referenced graphics;
   export to file "$(entity).tif";
}}
```

This rule tells FrameMaker that if it encounters a graphic element with an imported graphic file with a single TIFF facet, it should write that graphic to the file specified by `$(entity).tif`.

See also

Related rules	"convert referenced graphics" on page 372 "entity name is" on page 383 "notation is" on page 445 "specify size in" on page 456
Rules mentioned in synopses	"element" on page 376 "is fm graphic element" on page 422 "is fm equation element" on page 420 "anchored frame" on page 365 "equation" on page 385 "facet" on page 394 "writer" on page 467
General information on this topic	Chapter 19, "Translating Graphics and Equations"

external data entity reference

Use the `external data entity reference` rule to drop references to all external data entities. By default, FrameMaker stores such references as the marker text in non-element Entity Reference markers.

Synopsis and contexts

```
external data entity reference drop;
```

Arguments

None.

Details

- In markup, the values of general entity name attributes, such as those used with graphics, are not considered entity references. This rule does not affect how FrameMaker treats general entity name attributes. In XML such entity name attributes are the only way to reference non-parsed entities such as graphics.
- When you translate a markup document to FrameMaker, when the software encounters an external data entity reference such as:

  ```
  &door;
  ```

  it stores the reference as the text of a non-element DOC Entity Reference marker by default, with the following marker text:

  ```
  door
  ```

  When you translate a FrameMaker document to markup, it outputs the marker text of non-element DOC Entity Reference markers as entity references.

Examples

- To discard all external data entity references, use this rule:

  ```
  external data entity reference drop;
  ```

See also

Rules mentioned in synopses	"drop" on page 373
General information on this topic	Chapter 17, "Translating Entities and Processing Instructions"

external dtd

Use this rule to specify how an exported markup instance refers to the current structure application's DTD. By default, FrameMaker uses the name of the file containing the DTD as the system identifier in the external identifier for the DTD. The `external dtd` rule provides the software with a different external identifier. The different forms of the rule allow specification of a system identifier, public identifier, or both.

Synopsis and contexts

```
1. writer external dtd is system;
2. writer external dtd is system "sysid";
3. writer external dtd is public "pubid";
4. writer external dtd is public "pubid" "sysid";
```

Arguments

sysid	A system identifier.
pubid	A public identifier.

Details

- To use this rule, you must have a DTD specified for the current structure application. You define a structure application in the `structapps.fm` file.
- Use this rule when you export FrameMaker documents to markup documents.
- By default, FrameMaker does not reproduce the DTD in the document type declaration subset. Instead, it uses the filename of the DTD that was specified in the structure application to write a document type declaration of the form:

  `<!DOCTYPE` *doctype* `SYSTEM "`*fname*`" [. . .`

  where *doctype* is the document type name and *fname* is the DTD filename specified in the structure application. This rule allows you to specify different system and public identifiers.
- You cannot use the `external dtd` rule in the same read/write rules file as the `include dtd` rule.

Examples

- To specify a local DTD as an external DTD and include the path with the filename, you could use this rule:

  writer

  ```
  external dtd is
     system "/doc/dtds/manuals.dtd";
  ```

  Note that path information is system specific. The Windows platform requires two backslashes in paths in the rules file in order to translate as one backslash.
- To specify and locate the CALS DTD as an external DTD, you could use this rule:

  ```
  writer external dtd is
     public "-//USA-DOD//DTD MIL-M-38784B//EN"
     "/doc/dtds/cals.dtd";
  ```
- To specify just the CALS DTD as an external DTD using a public identifier, you could use this rule:

  ```
  writer external dtd is
     public “-//USA-DOD//DTD MIL-M-38784B//EN”;
  ```

  You could then specify the location of the DTD in the structure application using the `EntitiesLocation` element. A DTD is an entity in the strictest sense.

See also

Related rules	"include dtd" on page 408 "include sgml declaration" on page 409 "write structured document" on page 466 "write structured document instance only" on page 466
Rules mentioned in synopses	"writer" on page 467

facet

Use the `facet` rule only in an `element` rule for a graphic element, to provide information the software needs when writing a document containing graphics to markup. The `facet` rule applies only when a graphic element is an anchored frame containing only a single imported graphic object whose original file was in the *`facetname`* graphic format. Use this rule to specify information about the graphic file and/or entity declaration for instances of the graphic element.

Synopsis and contexts

```
element "gi" {
   is fm graphic element ["fmtag"];
   writer facet "facetname" subrule;
. . .}
```

Arguments

`gi`	A generic identifier.
`fmtag`	A FrameMaker element tag.
`facetname`	The name of the particular facet to which this rule applies.
`subrule`	A `facet` rule can have the following subrules: `convert referenced graphics`, page 372, tells the software to create new graphic files for imported graphic files with a single facet. `entity name is`, page 383, tells the software how to create the base name for the entity associated with this element type. `export to file`, page 389, tells the software the name to use for graphics it creates, and optionally, the graphic format to which it should convert. `notation is,` page 445, specifies the data content notation of the entity.

`specify size in`, page 456, specifies the units to use when writing the file.

Details

To specify all facets, use the keyword default for the facetname argument. For example:

```
element "pict" {
   is fm graphic element "Picture";
   writer {
      facet default {
      convert referenced graphics;
      export to file "$(entity).tif" as "TIFF";
      . . .
}}}
```

will convert every imported graphic file in the document to a TIFF file, no matter what its original facet was.

Examples

- By default, FrameMaker does not create a new graphic file for a graphic element that originated as an external entity, and was not modified by the user in any way. Assume you want the software to generate a graphic file for every imported TIFF file, whether it was modified or not. Then you could use this rule:

```
element "pict" {
   is fm graphic element "Picture";
   writer {
      facet "TIFF" {
         convert referenced graphics;
         export to file "$(entity).tif" as "TIFF";
}}}
```

See also

Related rules	"anchored frame" on page 365
	"convert referenced graphics" on page 372
	"equation" on page 385
Rules mentioned in synopses	"element" on page 376
	"is fm equation element" on page 420
	"is fm graphic element" on page 422
	"writer" on page 467
General information on this topic	Chapter 19, "Translating Graphics and Equations"

fm attribute

You use the `fm attribute` rule with the “drop” subrule to discard an attribute that you’ve defined for a FrameMaker element but that does not exist on the corresponding markup element. Read/write rules do not support double-byte characters, so you cannot use this rule to drop attributes with double-byte characters in their names.

Synopsis and contexts

```
1. fm attribute "attr" drop;

2. element "gi" {. . .
      fm attribute "attr" drop;
   . . .}
```

Arguments

attr	A FrameMaker attribute name.
gi	A generic identifier.

Examples

- Assume the element `chapter` exists in both the markup and FrameMaker representations of your documents. In FrameMaker, you use the `XRefLabel` attribute in formatting cross-references to this element. Since this attribute exists only for formatting purposes, you don’t want it in the markup document. To drop this attribute on export, use this rule:

  ```
  element "chapter" {
     is fm element;
     fm attribute "XRefLabel" drop;
  }
  ```

- If you use the `XRefLabel` attribute on many elements for the same purpose, you can discard it from all elements on export with this rule:

  ```
  fm attribute "XRefLabel" drop;
  ```

- If you want to keep the `XRefLabel` attribute on the `appendix` element, but drop it from all others, use these rules:

  ```
  element "appendix" {
     is fm element;
     attribute "xreflab" is fm attribute "XRefLabel";
  }
  fm attribute "XRefLabel" drop;
  ```

  Note that the order of these rules is not important. If you reversed them, the `XRefLabel` attribute would still be correctly interpreted for the `appendix` element, since that reference to the attribute is more specific. Note also that case is sensitive for fm attribute names.

See also

Related rules	“attribute” on page 367 “is fm attribute” on page 414
Rules mentioned in synopses	“element” on page 376 “drop” on page 373
General information on this topic	Chapter 16, “Translating Elements and Their Attributes”

fm element

Use the `fm element` rule to tell FrameMaker what to do on export with FrameMaker elements that do not correspond to markup elements. Read/write rules do not support double-byte characters, so you cannot use this rule to process elements with double-byte characters in their names.

Synopsis and contexts

1. **fm element** "*fmtag*" drop;
2. **fm element** "*fmtag*" unwrap;

Arguments

fmtag	A FrameMaker element tag.

Details

- Use this rule when you export FrameMaker documents to markup documents.
- If you use this rule, you may want to write a structure API client to handle the export of the element or to create it on import.
- The first version of this rule discards the FrameMaker element on export. The second version inserts the contents of *fmtag* in the corresponding markup document, but not *fmtag* itself.
- If you use this rule to unwrap FrameMaker cross-reference elements or system variable elements, those elements become text in the resulting markup document.

Examples

- If `Chapter Number` is a FrameMaker element that you want to discard on export, use this rule:

  ```
  fm element "Chapter Number" drop;
  ```

  If you use this rule and want to create this element on import, you need to write a structure API client.

- If `Modification Date` is a FrameMaker system variable element that you wish to translate to text on export to markup, use this rule:

```
fm element "Modification Date" unwrap;
```

See also

Related rules	"element" on page 376 "is fm element" on page 419
Rules mentioned in synopses	"drop" on page 373 "unwrap" on page 461
General information on this topic	Chapter 16, "Translating Elements and Their Attributes"

fm marker

On export, you use the `fm marker` rule to tell FrameMaker what to do with non-element markers other than markers of the type reserved for storing processing instructions, PI entities, and external data entities. (By default, Structure PI markers are reserved for processing instructions, and Entity Reference markers are reserved for external data entities.) In the absence of a rule to the contrary, the software creates processing instructions for non-element markers. You can also choose to discard them. Read/write rules do not support double-byte characters, so you cannot use this rule to process markers with double-byte characters in their names.

Synopsis and contexts

```
fm marker ["type1", . . ., "typen"] drop;
fm marker ["type1", . . ., "typen"] is processing instruction;
```

Arguments

type$_i$	The name of a FrameMaker marker type.

Details

- If *type*$_i$ is specified, this rule applies only to markers of that type.

  If no *type*$_i$ is specified, this rule applies to all non-element markers other than markers of the reserved type. For information on what the software does with the reserved marker type, see Chapter 22, "Translating Markers."

- You can have multiple occurrences of this rule in a rules file, to determine different treatment for different FrameMaker markers. You can have only one occurrence of the rule with no explicitly listed markers. A given marker type can be explicitly mentioned in only one occurrence of this rule.

- The order of `fm marker` rules is not important. A more specific occurrence of the rule always takes precedence over a more general occurrence. For example, the following rules:

```
fm marker "Index" is processing instruction;
fm marker drop;
```

  have the same effect, exporting only index markers as processing instructions, if they occur in this order:

```
fm marker drop;
fm marker "Index" is processing instruction;
```

Examples

- To discard all non-element markers, use this rule:

```
fm marker drop;
```

- To discard non-element conditional text markers but retain all others as processing instructions, use this rule:

```
fm marker "Conditional Text" drop;
```

- To retain only Index and Hypertext markers as processing instructions and drop all other non-element markers, use the following set of rules:

```
fm marker "Index", "Hypertext" is processing instruction;
fm marker drop;
```

See also

Related rules	“is fm marker element” on page 423
Rules mentioned in synopses	“drop” on page 373 “is processing instruction” on page 441
General information on this topic	Chapter 22, “Translating Markers”

fm property

You use the `fm property` rule to determine values for properties defined for certain types of FrameMaker constructs that you do not want to represent as markup attributes.

Synopsis and contexts

```
1. fm property prop value is “val”;

2. element "gi" {
       is fm type ["fmtag"];
       fm property prop value is val;.
   . . .}
```

Arguments

prop — A FrameMaker property. Possible properties are:

- For cross-reference elements: `cross-reference format`.
- For graphic and equation elements: `entity`, file, `dpi`, `import size`, `import by reference or copy`, `sideways`, `import angle`, `horizontal offset`, `vertical offset`, `position`, `baseline offset`, `near-side offset`, `alignment`, `cropped`, `floating`, `angle`, `width`, or `height`.
- For marker elements: `marker type` or `marker text`.
- For table elements: `column ruling`, `column widths`, `columns`, `page wide`, `row ruling`, `table border ruling`, or `table format`.
- For table cell elements: `column name`, `column number`, `column ruling`, `end column name`, `horizontal straddle`, `more rows`, `rotate`, `row ruling`, `span name`, `start column name`, or `vertical straddle`.
- For table row elements: `maximum height`, `minimum height`, `row type`, or `row ruling`.
- For CALS table colspecs: `cell alignment character`, `cell alignment offset`, `cell alignment type`, `column name`, `column number`, `column ruling`, `column width`, `row ruling`, or `vertical alignment`.
- For CALS table spanspecs: `cell alignment character`, `cell alignment offset`, `cell alignment type`, `column ruling`, `end column name`, `row ruling`, `span name`, `start column name`, or `vertical alignment`.

gi — A generic identifier.

fmtag — A FrameMaker element tag.

type — One of the following: `cross-reference element`, `graphic element`, `equation element`, `marker element`, `table element`, `table row element`, `table cell element`, `colspec`, or `spanspec`.

val — The keyword `default` or a string containing a legal value for the formatting property.

Details

- This rule applies only to an element corresponding to a cross-reference, graphic, equation, marker, table, or table part element.

- Some FrameMaker properties have no natural markup counterparts. If you choose to not translate such properties as markup attributes, a markup document will not contain information on appropriate values for these properties. In this situation, you can use the `fm property` rule to explicitly set property values when reading a markup document.
- This rule can be used either at the highest level to set a default or within an `element` rule to specify the translation of a property for a particular element.
- If you use this rule to set a property value explicitly, you cannot also have a markup attribute that corresponds to this property. For example, the following rule is erroneous:

```
element "tab2" {
   is fm table element;
   attribute "w" is fm property column widths;
   fm property column widths value is "1in 2in";
}
```

Examples

- To translate the markup element `table` to a FrameMaker table with two columns:

```
element "table" {
   is fm table element;
   fm property columns value is "2";
}
```

  On import to FrameMaker, the software creates the table as a 2-column table in FrameMaker.

- Assume you have a markup element `halfpage` that holds a 4.5 inch by 6.5 inch graphic object; it doesn't use an attribute to store the size information. You can translate this to a FrameMaker graphic as follows:

```
element "halfpage" {
   is fm graphic element;
   fm property width value is "6.5";
   fm property height value is "4.5";
}
```

See also

Related rules	"is fm property" on page 424
	"is fm property value" on page 426
General information on this topic	Chapter 19, "Translating Graphics and Equations"
	Chapter 18, "Translating Tables"
	Chapter 20, "Translating Cross-References"
	Chapter 22, "Translating Markers"

fm variable

On export, use the `fm variable` rule to tell FrameMaker what to do with certain variables. Use this rule if you do not want them translated to entities. Read/write rules do not support double-byte characters, so you cannot use this rule to process variables with double-byte characters in their names.

Synopsis and contexts

fm variable ["var_1", . . ., "var_n"] drop;

Arguments

var_i	The name of a FrameMaker variable.

Details

- Use this rule when you export FrameMaker documents to markup documents. It applies only to non-element variables, not to system variable elements.
- If var_i is specified, this rule applies only to that variable. If no var_i is specified, this rule applies to all variables.
- If you use this rule, you may want to write a structure API client to handle the export of variables or to create variables on import.
- You can have multiple occurrences of this rule in a rules document to determine different treatment for different FrameMaker variables. You can have only one occurrence of the rule with no explicitly listed variables. A given variable can be explicitly mentioned in only one occurrence of this rule.

Examples

- To translate the FrameMaker variables `Licensor` and `Product` as entities and discard all other variables, use these rules:

```
entity "licensor" is fm variable;
entity "product" is fm variable;
fm variable drop;
```

See also

Related rules	"is fm system variable element" on page 433
General information on this topic	Chapter 17, "Translating Entities and Processing Instructions" Chapter 21, "Translating Variables and System Variable Elements" *Structure Import/Export API Programmer's Guide*

fm version

The `fm version` rule specifies the version of the product being run. It is required and must be the first rule in all rules documents. If you create your rules document with the New Read/Write Rules command, this rule automatically appears in the document.

Synopsis and contexts

```
fm version is "7.1";
```

Arguments

None.

Details

- Note that you would use the string `"7.1"` in this rule even though the product version may be an incremental release above 7.1, such as 7.1.1.

See also

General information on this topic — Chapter 14, “Read/Write Rules and Their Syntax”

generate book

Use the `generate book` subrule of a highest-level `reader` rule to specify whether FrameMaker should use elements or processing instructions to indicate where in a markup document to start a book and its components in the corresponding FrameMaker book.

Synopsis and contexts

```
1. reader generate book
      use processing instructions;

2. reader generate book
   {
      put element "gi1" in file ["fname1"];
      . . .
      put element "giM" in file ["fnameM"];
   }

3. reader generate book [for doctype "dt1" [, . . . "dtN"]]
   {
      put element "gi1" in file ["fname1"];
      . . .
      put element "giM" in file ["fnameM"];
   }
```

Arguments

dt_i	A document type name.

gi$_j$	A generic identifier.
fname$_j$	A filename for the book component. FrameMaker adds a counter to the name (before the suffix if there is one) as needed, to generate a unique filename. You can use the `$(bookname)` variable to base the component's filename on the book filename (excluding any suffix). If you do not supply this argument, the filename is *gi$_j$*`.doc`.

Details

- By default, when reading a markup document into FrameMaker, the software uses the `<?FM book ?>` and `<?FM document ?>` processing instructions to indicate the start of a book and of its components. The following rule confirms this default behavior:

```
reader generate book
   use processing instructions;
```

- Your DTD may be defined so that you can use elements to indicate the start of a book and its components. When you use the second form of the `generate book` rule, FrameMaker creates a book for every markup document you translate. When you use the third form of the `generate book` rule, it creates a book only for markup documents whose DTD specifies the document type you've listed in the rule. If you have a markup document with a different document type, FrameMaker translates that document as a single FrameMaker document, even if it contains elements referenced in `put element` rules. For example, assume you have this rule:

```
reader generate book for doctype "manual"
   put element "chapter" in file;
```

  If you translate a markup document whose highest-level element is `report`, that document becomes a single FrameMaker document, even if it contains `chapter` descendant elements.

- When it encounters one of the *gi$_j$* elements specified in a `put element` subrule, FrameMaker starts a new book component. Since the software does not allow an element to be broken across files, it places the entire *gi$_j$* element in the same file, even if another element appears that you've said should start a new file. To illustrate, assume the `section` element can occur either within or outside of a `chapter` element and you have this rule:

```
reader generate book {
   put element "chapter" in file;
   put element "section" in file;
}
```

  When FrameMaker encounters a `chapter` element, it starts a new file. If it encounters a `section` element as a child of that `chapter` element, it does not start a new file. It continues with the file started by the `chapter` element. On the other hand, if the software encounters a `section` element outside a `chapter` element it does start a new file for it.

- There are a couple of points to consider when dividing a markup document into book components:
  - Every FrameMaker document must contain exactly one highest-level element. That is, there cannot be two elements in a single file that do not have an ancestor element in the same file.
  - A book element can contain substructure but cannot directly contain text. That is, child elements that can contain text must occur in separate files.

  Assume you have this rule:

  ```
  reader generate book
     put element "chapter" in file;
  ```

  And you have a markup document with the following element structure:

  ```
  <manual>
  <chapter>
  <head>Introduction</head>
  . . .
  </chapter>
  <appendix>
  <head>The final word</head>
  . . .
  </appendix>
  </manual>
  ```

  When FrameMaker translates this document, it creates a book with `manual` as the highest-level element in the book file. When it encounters the `chapter` element, the software starts a new file for that element. When it encounters the `appendix` element, FrameMaker flags an error, because your rules have not told it what to do with this element. It cannot put the element in the same file as the preceding `chapter` element, because that would create two highest-level elements in the same file. It also cannot put the `appendix` element in the book file, because it contains text.

- By default, when it writes a FrameMaker book to markup, the software writes `<?FM book ?>` and `<?FM document ?>` processing instructions for the book and book components. It does this even if you use the `generate book` rule to have particular elements specify book components when reading a markup document. If you do not want FrameMaker to output these processing instructions, use the `do not output book processing instructions`, which is described on page 447.

Examples

- If you know that a markup document should always correspond to a FrameMaker book and that individual files in the book should start when the document reaches a `toc` or `chapter` element, you can use this rule:

```
reader generate book {
   put element "toc" in file;
   put element "chapter" in file "ch.doc";
}
```

  With this rule, FrameMaker creates a book for each markup document. In a markup document, FrameMaker starts a new book component when it encounters a `toc` or `chapter` element. For the first `toc` element, FrameMaker uses the filename `toc1` unless a file of that name already exists in the directory it is using. It continues that book component until it encounters either another `toc` element or a `chapter` element. At that point, it starts a new book component. It tries to put the first `chapter` element in a file called `ch1.doc`.

- Assume that a markup document whose highest-level element is either `manual` or `book` should correspond to a FrameMaker book and any other markup document should correspond to an individual FrameMaker document. Further assume that the books created from `manual` and `book` elements should have new files for each instance of the elements `chapter`, `front`, or `toc`. To accomplish all this, you can use this rule:

```
reader generate book for doctype "manual", "book"
{
   put element "chapter" in file "ch.doc";
   put element "front" in file;
   put element "toc" in file "$(bookname).toc";
}
```

  With this rule, FrameMaker asks you for a name for the book file if you open a markup document with `manual` as its document type. If you specify `myfile.bk` as its name, and the document contains two `chapter` elements, one `front` element, and one `toc` element, FrameMaker creates the following files: `myfile.bk`, `ch1.doc`, `ch2.doc`, `front`, and `myfile.toc`.

See also

Related rules	“output book processing instructions” on page 447
General information on this topic	Chapter 24, “Processing Multiple Files as Books”

implied value is

Use the `implied value is` rule to specify default attribute values in your EDD to correspond with imported elements that specify no value for the attribute. For example, assume your DTD declares an element named `list`, which has an attribute named `style`

defined as <!ATTLIST list style (bul | num) #IMPLIED>. For importing the DTD, you can use this rule to set up a default value in the EDD for the Style attribute of the List element. Then, if you import a list element that has no value for style, this default attribute value will be used for formatting purposes. Also, when you export the EDD, the DTD will declare the style attribute for the list element as #IMPLIED.

Synopsis and contexts

```
1. attribute "attr" {. . .
      implied value is "val";
   . . .}

2. element "gi" {. . .
      attribute "attr" {. . .
         implied value is "val";
   . . .} . . .}
```

Arguments

attr	The name of an impliable attribute in markup.
val	A value to use for the *attr* attribute.
gi	A generic identifier.

Details

- This rule is for importing DTDs and exporting EDDs. In FrameMaker, a default attribute value can only be specified in the EDD, so this rule has no effect when importing a markup instance or exporting a FrameMaker document.
- This rule specifically does not supply an attribute value for an element that has no value in the markup instance. It only sets up a default attribute value in the EDD. This default value can be used for formatting by attributes. When you export the document, FrameMaker will not add a value for the attribute to the element's start tag.
- The rule can be used in a highest-level attribute rule to specify the value to use for that attribute in any element. Alternatively, it can be used in an attribute rule within an element rule to specify the value for that element only.

Examples

- Assume you have these declarations for a markup element used for cross-references:

```
<!ELEMENT xref - o EMPTY>
<!ATTLIST xref
    id IDREF #IMPLIED
    format CDATA #IMPLIED>
```

And you have this rule:

```
element "xref" {
   is fm cross-reference element;
   attribute "format" {
      is fm property cross-reference format;
      implied value is "Page";
}}
```

When FrameMaker encounters an instance of the `xref` element in a markup document and that instance doesn't have a value for the `format` attribute, the software use the Page cross-reference format for the cross-reference in the FrameMaker document.

See also

Related rules	"value" on page 464
Rules mentioned in synopses	"attribute" on page 367 "element" on page 376
General information on this topic	Chapter 16, "Translating Elements and Their Attributes" "Default value" on page 176

include dtd

By default, when creating a markup document, FrameMaker includes in the document type definition an external identifier that refers to the DTD file. Therefore, it does not include a copy of actual declarations in the document type declaration subset. The `include dtd` rule tells FrameMaker to do so.

Synopsis and contexts

```
writer [do not] include dtd;
```

Arguments

None.

Details

- You use this rule when you export FrameMaker documents to markup documents. If this rule is specified, FrameMaker does not generate an external identifier in the `DOCTYPE` declaration.
- To confirm the default behavior, you can use the opposite rule:

  ```
  writer do not include dtd;
  ```

- The `include dtd` rule and the `external dtd` rule are mutually exclusive. That is, you cannot use both of these rules in the same read/write rules file. (If you try to put both of these rules in the same file, you will get an alert.) Also, the `include dtd` rule and the `write structure document instance only` rule are mutually exclusive.

- To write an entire markup document, including a DTD and (for SGML) an SGML declaration with the document instance, you must use the following rules:

```
writer {
   include sgml declaration;
   include dtd;
}
```

- This rule uses the DTD that is specified in the current structure application. If that DTD includes references to external files, this rule does not expand those references as it writes out the DTD. Instead, it writes out the references as they appear in the parent DTD file.
- You can use this rule to translate the EDD from the current document as an a DTD that is written in the markup document. To do this, use the `include dtd` rule, but use a structure application that does not specify a DTD in its definition. Be warned that if your document uses the CALS table model, the resulting DTD may be incorrect.

Examples

- If your document type declarations are in a file called `report.dtd`, then by default FrameMaker includes this document type declaration in the document it creates on export:

```
<!DOCTYPE report SYSTEM "report.dtd" [
. . . more declarations specific to this document instance . . .
]>
```

If you specify the `include dtd` rule, then FrameMaker includes this document type declaration in the document it creates:

```
<!DOCTYPE report [
. . . declarations specific to this document instance . . .
. . . contents of the file, report.dtd . . .
]>
```

See also

Related rules

"external dtd" on page 392
"include sgml declaration," next
"write structured document" on page 466
"write structured document instance only" on page 466

include sgml declaration

By default, FrameMaker does not include an SGML declaration in a generated SGML document. The `sgml declaration` rule tells FrameMaker to include one. The SGML

declaration is copied from the file in the associated application subset. To see the default SGML declaration used by FrameMaker, see Appendix D, "SGML Declaration."

XML: This read/write rule is for SGML-only.

Synopsis and contexts

```
writer [do not] include sgml declaration;
```

Arguments

None.

Details

- To confirm the default behavior, you can use the opposite rule:

  ```
  writer do not include sgml declaration;
  ```

- You cannot use the `include sgml declaration` rule in the same read/write rules file as the `write sgml document instance only` rule. Note that using both rules in the same rules file doesn't give an error. Also, "write sgml document instance only" takes priority, regardless of order.

- To write an entire SGML document, including an SGML DTD and SGML declaration with the document instance, you must use the following rules:

  ```
  writer {
     include sgml declaration;
     include dtd;
  }
  ```

See also

Related rules

"external dtd" on page 392
"include dtd," (the previous section)
"write structured document" on page 466
"write structured document instance only" on page 466

insert table part element

You use the `insert table` *part* `element` rule when creating a FrameMaker table element on import of a markup document. This rule tells FrameMaker to create a table part of the designated type, even if the markup document does not contain content for that table part.

Synopsis and contexts

```
element "gi" {. . .
   is fm table element ["fmtag1"];
   reader insert table part element ["fmtag2"];
. . .}
```

Arguments

gi	A generic identifier.
$fmtag_1$	A FrameMaker element tag for a table element.
part	One of the keywords: `title`, `heading`, or `footing`.
$fmtag_2$	A FrameMaker element tag for a table part element.

Details

- By default, as the last step in creating a table element when reading a markup document, FrameMaker discards parts of the table that have no content, even if the general rule for the element requires that table part. (Your EDD may supply the content, for example, by using format rules that specify a prefix for the element.) If you do not want FrameMaker to remove the table part element with no content, OR if you want FrameMaker to create a table part element for you when the markup instance does not contain this element, use the `insert table` *part* `element` rule.

Examples

- Assume you have a markup element `statetab`, which you represent as a 3-column table in FrameMaker, with the same table headings everywhere it occurs. You use formatting rules in the EDD to specify the table headings. In this situation, the markup document does not include information that corresponds to the table headings, so you want the software to add the table heading element when reading such a markup instance and drop it when exporting a FrameMaker document to markup. Suppose your DTD has these declarations:

```
<!ELEMENT statetab - - ((state, pop, income)+)>
<!ELEMENT state - - (#PCDATA)>
<!ELEMENT pop - - (#PCDATA)>
<!ELEMENT income - - (#PCDATA)>
```

  and your EDD has these FrameMaker element definitions:

  Element (Table): State Table
  General rule: State Head, State Body
  Text format rules
  1. In all contexts.
  Use paragraph format: TableCell

  Element (Table Heading): State Head
  General rule: State Head Row
  Text format rules
  1. In all contexts.
  Default font properties
  Weight: Bold

Element (Table Row): State Head Row
General rule: Label

Element (Table Cell): Label
General rule: <EMPTY>
Text format rules
1. If context is: {first}
Numbering properties
Autonumber format: State
Else if context is: {last}
Numbering properties
Autonumber format: Household Income
Else
Numbering properties
Autonumber format: Population

Element (Table Body): State Body
General rule: State Row+

Element (Table Row): State Row
General rule: State, Income, Population

Element (Table Cell): State
General rule: <TEXT>

Element (Table Cell): Income
General rule: <TEXT>

Element (Table Cell): Population
General rule: <TEXT>

Note that the Label element provides the text for the column headings.

You could use these rules:

```
element "statetab" {
   is fm table element "State Table";
   fm property columns value is "3";
   reader insert table heading element "State Head";
}

element "state" {
   is fm table cell element;
   fm property column number value is "1";
   fm property row type value is "Body";
}

element "income" is fm table cell element;

element "pop" is fm table cell element "Population";
```

```
fm element "State Head" drop;
fm element "State Body" unwrap;
fm element "State Row" unwrap;
```

To convert the following instance to the desired FrameMaker document:

```
<statetab>
<state>Georgia</state><pop>15,000,000</pop><income>25,000</
income>
<state>Mississippi</state><pop>8,000,000</pop><income>18,000</
income>
</statetab>
```

The first rule identifies `statetab` as a 3-column table element and tells it to always create a heading element for an occurrence of this `statetab`.

The second rule identifies `state` as a table cell that must always occur in the first column of a body row. This ensures that FrameMaker starts a new table row whenever it encounters a `state` element.

The other element rules identify other elements used as table cells. The `fm element drop` rule causes the software to drop the element that was created by the software per the `insert element` rule so that it does not appear in the markup. Note also that it is necessary for the software to have a `tablerow` element and a `tablebody` element in its table structure. However, these do not appear in the markup document. The software creates such necessary elements by default.Since they do not correspond to markup elements, they are unwrapped on export to markup—not dropped, because they would lose the contents of the entire table.

See also

General information on this topic	Chapter 18, “Translating Tables”

is fm attribute

Use the `is fm attribute` rule to specify that a markup attribute translates to a FrameMaker attribute. The optional parts of this rule allow you to have the software make several changes to the attribute during translation. Read/write rules do not support double-byte characters, so you cannot use this rule to process attributes with double-byte characters in their names.

Synopsis and contexts

```
1. [structdv] attribute "sgmlattr" {. . .
      is fm [read-only] [fmtype] attribute
         ["fmattr"] [range from low to high];
   . . .}

2. element "gi" {. . .
      [structdv] attribute "sgmlattr"
         is fm [read-only] [fmtype] attribute
            ["fmattr"] [range from low to high];
   . . .}
```

Arguments

structdv	An optional markup declared value. Legal values are: `cdata`, `name`, `names`, `nmtoken`, `nmtokens`, `number`, `numbers`, `nutoken`, `nutokens`, `entity`, `entities`, `notation`, `id`, `idref`, `idrefs`, and `group`.
sgmlattr	A markup attribute name.
fmtype	A FrameMaker attribute type. Legal values are: String, Strings, Integer, Integers, Real, Reals, UniqueID, IDReference, IDReferences, and Choice.
fmattr	A FrameMaker attribute name.
low	A number, indicating the low end of a numeric range.
high	A number, indicating the high end of a numeric range.

Details

- You can use the `is fm attribute` rule in a highest-level `attribute` rule to specify the translation of that attribute in all elements for which it is defined. Or you can use it in an `attribute` subrule in an `element` rule to specify the translation of the `attribute` in only that element.
- You may want some markup attributes to become FrameMaker properties. If so, you cannot also import them as FrameMaker attributes. For information on the defined FrameMaker properties, see "is fm property" on page 424.

- To specify only that the attribute is an attribute in both representations, use this version:

```
attribute "sgmlattr" is fm attribute;
```

- To also rename it during translation, use this version:

```
attribute "sgmlattr" is fm attribute "fmattr";
```

- To specify that the FrameMaker attribute is read-only—that is, that an end user cannot change the attribute's value—use this version:

```
attribute "sgmlattr" is fm read-only attribute;
```

- To specify that an attribute that takes numeric values can have values only in a particular range, use this version:

```
attribute "sgmlattr" is fm attribute range from low to high;
```

- To specify that a markup attribute with a particular declared value translates to a FrameMaker attribute of a type other than the default translation, use this version:

```
structdv attribute "sgmlattr" is fm fmtype attribute;
```

- Note that you can use more than one of the optional pieces of the `is fm attribute` rule at the same time. For example, you can both rename an attribute and state that it is read-only by using this version:

```
attribute "sgmlattr" is fm read-only attribute "fmattr";
```

Examples

- To translate the markup `sec` attribute to the FrameMaker `SecurityRanking` attribute in all elements in which it occurs, use this rule:

```
attribute "sec" is fm attribute "SecurityRanking";
```

- To translate the markup `sec` attribute to the FrameMaker `SecurityRanking` attribute in most elements in which it occurs, but to change it to the `Section` attribute in the `BookPart` element, use these rules:

```
element "BookPart"
   attribute "sec" is fm attribute "Section";

attribute "sec" is fm attribute "SecurityRanking";
```

- Assume you have a markup attribute named `perc` with a declared value of `CDATA`, and assume you know that this attribute always has values that are integers in the range from 0 to 100. You can translate the `perc` attribute to the `Percentage` attribute with this rule:

```
cdata attribute "perc"
   is fm integer attribute "Percentage" range from 0 to 100;
```

- Assume that a markup element has an attribute with declared value name and that the attribute has a defined set of allowable values. You can translate that attribute and some of its possible values with the following rule:

```
element "fish" {
   name attribute "loc" {
      is fm choice attribute "CommonLocation";
      value "micro" is fm value "Micronesia";
      value "galap" is fm value "Galapagos Islands";
      value "png" is fm value "Papua New Guinea";
}}
```

See also

Related rules	"fm attribute" on page 396
Rules mentioned in synopses	"attribute" on page 367 "element" on page 376
General information on this topic	Chapter 16, "Translating Elements and Their Attributes"

is fm char

For SGML, use the `is fm char` rule to translate an SGML SDATA entity to a single character in FrameMaker. For XML, use this rule to translate an internal entity to a single character in FrameMaker.

Synopsis and contexts

1. `entity "`*`ename`*`" **is fm char** `*`ch`*` [in "`*`fmchartag`*`"];`

2. `reader entity "`*`ename`*`" **is fm char** `*`ch`*` [in "`*`fmchartag`*`"];`

Arguments

ename	An entity name.
ch	A one-character string or a numeric character code (specified using the syntax for an octal, hexadecimal, or decimal number described in "Strings and constants" on page 212). Note that if the desired character is a digit or a white-space character, you must enter it as a numeric character code.
fmchartag	A FrameMaker character format tag. Note that the character format must use a non-standard font family such as Symbol or Zapf Dingbats for this argument to take effect.

Details

- For SGML, instead of using this rule to translate an SDATA entity, you can use a parameter literal of a particular form. For information on how to do so, see "Translating SDATA entities as special characters in FrameMaker" on page 260.
- For XML, SDATA entities are not allowed. This rule translates internal entities to FrameMaker characters, and it translates FrameMaker to internal entities.
- You can use the `is fm char` rule within an `entity` rule at the highest level to have the translation occur in both directions. Or you can put the `entity` rule inside a `reader` rule to have the translation occur only when reading a markup document into FrameMaker. For example, your SGML document might use a `period` entity for entering some instances of the period character in your SGML document. If you use this rule:

  ```
  entity "period" is fm char ".";
  ```

  then the entity references for period in the instance are translated correctly to the period character in FrameMaker. But on export, all periods in the document become references to the `period` entity (which is not likely what you had in mind). To have the period entities read correctly when importing an instance, but have periods remain the period character on export, use this version of the rule:

  ```
  reader
     entity "period" is fm char ".";
  ```

- Without the `in` clause, the software translates the entity using the default character format of the enclosing paragraph element. Frequently, however, special characters require a font change. In these cases, you use the `in` clause.
- For SGML, DTDs frequently use the entity sets defined in Annex D of the SGML Standard, often called ISO public entity sets, for providing commonly used special characters. FrameMaker includes copies of these entity sets and provides rules to handle them for your application. For information on how FrameMaker supports ISO public entities, see Appendix F, "ISO Public Entities."

Examples

- To translate the SDATA entity `sum` as the mathematical summation sign in the Symbol font (Σ), you could use either of these rules in your rules document:

  ```
  entity "sum" is fm char "S" in "Symbol";

  entity "sum" is fm char "\x53" in "Symbol";

  entity "sum" is fm char 0x53 in "Symbol";
  ```

  If FrameMaker encounters a reference to the `summation` entity when importing a markup document, it replaces the reference with Σ (assuming your FrameMaker template defines the Symbol character format appropriately and the entity is declared in the DTD). If the software encounters Σ when exporting an document, it generates a reference to the `summation` entity (assuming the Symbol character format is defined appropriately

and applied to the character, and that the DTD for your application has an entity declaration for “sum”).

- To translate both the `thin` and `en` internal entity references in an XML instance to en spaces in FrameMaker and to write all en spaces as an `en` entity reference, use these rules:

```
entity "en" is fm char 0x13;
reader entity "thin" is fm char 0x13;
```

See also

Rules mentioned in synopses — “entity” on page 381

General information on this topic — Chapter 17, “Translating Entities and Processing Instructions”

is fm cross-reference element

Use the `is fm cross-reference element` rule to identify a markup element that translates to a cross-reference element in FrameMaker. You can choose either to have the same name in both representations or to change the name during translation. The markup element should have an attribute of type IDREF and declared content of EMPTY. Read/write rules do not support double-byte characters, so you cannot use this rule to process elements with double-byte characters in their names.

Synopsis and contexts

```
element "gi" {. . .
   is fm cross-reference element ["fmtag"];
. . .}
```

Arguments

gi	A generic identifier.
fmtag	A FrameMaker element tag.

Details

- If you use the `is fm cross-reference element` rule, the other subrules of the `element` rule that you can use for that markup element are as follows:

  `attribute`, page 367, specifies what to do with a markup element’s attributes.

  `fm attribute`, page 396, specifies what to do with attributes present in the FrameMaker representation of the element but not in the markup representation.

  `fm property`, page 399, specifies what to do with FrameMaker properties associated with the element.

  `reader drop content`, page 375, specifies that the content but not the structure of an element should be discarded on import of a markup document.

Examples

- To have the markup element `xref` become the FrameMaker cross-reference element `Xref`, use this rule:

```
element "xref" is fm cross-reference element;
```

- To have it become the FrameMaker cross-reference element `CrossRef`, use this rule:

```
element "xref" is fm cross-reference element "CrossRef";
```

See also

Rules mentioned in synopses — "element" on page 376

General information on this topic — Chapter 20, "Translating Cross-References"

is fm element

If you do not specify a value for *fmtag*, the `is fm element` rule specifies only that a markup element remains an element in FrameMaker. This is the default behavior. With a value for *fmtag*, this rule changes the element name when it is translated between markup and FrameMaker.

Synopsis and contexts

```
element "gi" {. . .
   is fm element ["fmtag"];
. . .}
```

Arguments

gi	A generic identifier.
fmtag	A FrameMaker element tag.

Details

- If you use the `is fm element` rule, the other subrules of the `element` rule that you can use for that markup element are as follows:

  `attribute`, page 367, specifies what to do with a markup element's attributes.

  `fm attribute`, page 396, specifies what to do with attributes present in the FrameMaker representation of the element but not in the markup representation.

  `fm property`, page 399, specifies what to do with FrameMaker properties associated with the element.

  `reader drop content`, page 375, specifies that the content but not the structure of an element should be discarded on import of a markup document.

  `writer drop content`, page 375, specifies that the content but not the structure of an element should be discarded on export of a FrameMaker document.

Examples

- To translate the markup element `par` to the FrameMaker element `Paragraph`, use this rule:

```
element "par" is fm element "Paragraph";
```

See also

Rules mentioned in synopses — "element" on page 376

General information on this topic — Chapter 16, "Translating Elements and Their Attributes"

is fm equation element

Use the `is fm equation element` rule to identify a markup element that translates to an equation element in FrameMaker. You can choose either to have the same name in both representations or to change the name during translation. Read/write rules do not support double-byte characters, so you cannot use this rule to process elements with double-byte characters in their names.

Synopsis and contexts

```
element "gi" {. . .
   is fm equation element ["fmtag"];
. . .}
```

Arguments

gi	A generic identifier.
fmtag	A FrameMaker element tag.

Details

- If you use this rule, the other subrules of the `element` rule that you can use for the same markup element are as follows:

  `attribute`, page 367, specifies what to do with a markup element's attributes.

  `fm attribute`, page 396, specifies what to do with attributes present in the FrameMaker representation of the element but not in the markup representation.

  `fm property`, page 399, specifies what to do with FrameMaker properties associated with the element.

  `writer equation`, page 385, tells FrameMaker what to do with equation elements.

Examples

- To have FrameMaker equation element `Eqn` become the markup element `eqn`, use this rule:

```
element "eqn" is fm equation element;
```

- To have FrameMaker equation element `Equation` become the markup element `eqn`, use this rule:

```
element "eqn" is fm equation element "Equation";
```

See also

Related rules	"is fm graphic element" on page 422
Rules mentioned in synopses	"element" on page 376
General information on this topic	Chapter 19, "Translating Graphics and Equations"

is fm footnote element

Use the `is fm footnote element` rule to identify a markup element that translates to a footnote element in FrameMaker. You can choose either to have the same name in both representations or to change the name during translation. Read/write rules do not support double-byte characters, so you cannot use this rule to process elements with double-byte characters in their names.

Synopsis and contexts

```
element "gi" {. . .
   is fm footnote element ["fmtag"];
. . .}
```

Arguments

gi	A generic identifier.
fmtag	A FrameMaker element tag.

Details

- If you use this rule, the other subrules of the `element` rule that you can use for the same markup element are as follows:

  `attribute`, page 367, specifies what to do with a markup element's attributes.

  `fm attribute`, page 396, specifies what to do with attributes present in the FrameMaker representation of the element but not in the markup representation.

Examples

- To translate the markup element `fn` to the `Fn` footnote element in FrameMaker, use this rule:

```
element "fn" is fm footnote element;
```

- To translate it to the `Footnote` footnote element, use this rule:

```
element "fn" is fm footnote element "Footnote";
```

See also

Rules mentioned in synopses — "element" on page 376

General information on this topic — Chapter 16, "Translating Elements and Their Attributes"

is fm graphic element

Use the `is fm graphic element` rule to identify a markup element that translates to a graphic element in FrameMaker. You can choose either to have the same name in both representations or to change the name during translation. Read/write rules do not support double-byte characters, so you cannot use this rule to process elements with double-byte characters in their names.

Synopsis and contexts

```
element "gi" {. . .
   is fm graphic element ["fmtag"];
. . .}
```

Arguments

gi	A generic identifier.
fmtag	A FrameMaker element tag.

Details

- If you use this rule, the other subrules of the `element` rule that you can use for the same markup element are as follows:

  `attribute`, page 367, specifies what to do with a markup element's attributes.

  `fm attribute`, page 396, specifies what to do with attributes present in the FrameMaker representation of the element but not in the markup representation.

  `fm property`, page 399, specifies what to do with FrameMaker properties associated with the element.

  `writer anchored frame`, page 365, tells FrameMaker what to do with graphic elements other than those with a single non-internal FrameMaker facet.

  `writer facet`, page 394, tells FrameMaker what to do with an imported graphic element that has a single non-internal FrameMaker facet.

Examples

- To translate the markup element `pict` to the `Pict` graphic element in FrameMaker, use this rule:

```
element "pict" is fm graphic element;
```

- To translate it to the `Picture` graphic element, use this rule:

```
element "pict" is fm graphic element "Picture";
```

See also

Related rules	"is fm equation element" on page 420
Rules mentioned in synopses	"element" on page 376
General information on this topic	Chapter 19, "Translating Graphics and Equations"

is fm marker element

Use the `is fm marker element` rule to identify a markup element that translates to a marker element in FrameMaker. You can choose either to have the same name in both representations or to change the name during translation. Read/write rules do not support double-byte characters, so you cannot use this rule to process elements with double-byte characters in their names.

Synopsis and contexts

```
element "gi" {. . .
   is fm marker element ["fmtag"];
. . .}
```

Arguments

gi	A generic identifier.
fmtag	A FrameMaker element tag.

Details

- If you use this rule, the other subrules of the `element` rule that you can use for the same markup element are as follows:

  `attribute`, page 367, specifies what to do with a markup element's attributes.

  `fm attribute`, page 396, specifies what to do with attributes present in the FrameMaker representation of the element but not in the markup representation.

  `fm property`, page 399, specifies what to do with FrameMaker properties associated with the element.

  `marker text is`, page 444, specifies whether the text of a FrameMaker marker element should be element content or an attribute value in markup.

Examples

- To translate the markup element `m` to the `M` marker element in FrameMaker, use this rule:

```
element "m" is fm marker element;
```

- To translate it to the `Marker` marker element, use this rule:

```
element "m" is fm marker element "Marker";
```

See also

Related rules	"marker text is" on page 444 "fm marker" on page 398
Rules mentioned in synopses	"element" on page 376
General information on this topic	Chapter 22, "Translating Markers"

is fm property

Use the `is fm property` rule to translate a markup attribute to a FrameMaker property. This rule can apply in a highest-level `attribute` rule to set a default. Or it can apply within an `element` rule for a table, table part, marker, cross-reference, graphic, or equation element, to set the property only for that element. Read/write rules do not support double-byte characters, so you cannot use this rule to process attributes with double-byte characters in their names.

Synopsis and contexts

```
1. attribute "attr" {. . .
      is fm property prop;
   . . .}
2. element "gi" {. . .
      attribute "attr" {. . .
         is fm property prop;
   . . .} . . .}
```

Arguments

attr — The name of a markup attribute.

prop — A FrameMaker property. Possible properties are:

- For cross-reference elements: `cross-reference format`, `cross-reference id`.
- For graphic and equation elements: `entity`, `file`, `dpi`, `import size`, `import by reference or copy`, `sideways`, `import angle`, `horizontal offset`, `vertical offset`, `position`, `baseline offset`, `near-side offset`, `alignment`, `cropped`, `floating`, `angle`, `width`, or `height`.
- For marker elements: `marker type` or `marker text`.

- For table elements: `column ruling`, `column widths`, `columns`, `page wide`, `row ruling`, `table border ruling`, or `table format`.
- For table cell elements: `column name`, `column number`, `column ruling`, `end column name`, `horizontal straddle`, `more rows`, `rotate`, `row ruling`, `span name`, `start column name`, or `vertical straddle`.
- For table row elements: `maximum height`, `minimum height`, `row type`, or `row ruling`.
- For CALS table colspecs: `cell alignment character`, `cell alignment offset`, `cell alignment type`, `column name`, `column number`, `column ruling`, `column width`, `row ruling`, or `vertical alignment`.
- For CALS table spanspecs: `cell alignment character`, `cell alignment offset`, `cell alignment type`, `column ruling`, `end column name`, `row ruling`, `span name`, `start column name`, or `vertical alignment`.

gi — A generic identifier.

Details

- If you use the `is fm property` rule to translate a markup attribute to a FrameMaker property, the markup attribute does not also appear as a FrameMaker attribute.
- If you use this rule in a highest-level `attribute` rule, it applies only to elements that have that attribute and are of the appropriate type. For example, if you have these declarations:

  ```
  <!ATTLIST (graphic | table) w CDATA #IMPLIED>
  ```

  and these rules:

  ```
  attribute "w" is fm property width;
  element "graphic" is fm graphic element;
  element "table" is fm table element;
  ```

  the `w` attribute becomes the `width` property of the `graphic` element but remains an attribute for the `table` element, since tables do not have a `width` property. If you intended `w` to be the column width for tables, you should use these rules:

  ```
  element "graphic" {
     is fm graphic element;
     attribute "w" is fm property width;
  }
  ```

```
element "table" {
   is fm table element;
   attribute "w" is fm property column width;
}
```

Examples

- The markup attribute `w` may be used for multiple elements to represent the width of a table's columns. To translate it to the FrameMaker property `column width`:

  ```
  attribute "w" is fm property column width;
  ```

- To translate the attribute `form` to the cross-reference formatting property `cross-reference format` for the element `xref`:

  ```
  element "xref" {
     is fm cross-reference element;
     attribute "form" is fm property cross-reference format;
  }
  ```

See also

Related rules	"fm property" on page 399 "is fm property value," next
Rules mentioned in synopses	"element" on page 376 "attribute" on page 367
General information on this topic	"Formatting properties for tables" on page 276 "Anchored frame properties" on page 302 "Other graphic properties" on page 304 Chapter 22, "Translating Markers" Chapter 20, "Translating Cross-References"

is fm property value

Use the `is fm property value` rule when a markup attribute has a name token group as its declared value and you want to rename the individual name tokens when translating to and from FrameMaker property values. Read/write rules do not support double-byte characters, so you cannot use this rule to process attributes with double-byte characters in their names.

Synopsis and contexts

```
1. value "token" is fm property value propval;

2. attribute "attr" {. . .
      value "token" is fm property value propval;
   . . .}
```

```
3. element "gi" {. . .
      attribute "attr" {. . .
         value "token" is fm property value propval;
   . . .} . . .}
```

Arguments

token	A token in a name token group.
propval	A defined FrameMaker property value.
attr	The name of a markup attribute.
gi	A generic identifier.

Details

- This rule can be used at the highest level to set a default, or within an `attribute` rule.
- Use this rule when the corresponding markup attribute translates to a property in FrameMaker. If the markup attribute translates to a choice attribute instead, you need to use the `is fm value` rule to specify the correspondence between markup tokens and FrameMaker attribute choices.
- When using this rule, remember that markup does not permit a token to appear in the declared value of more than one attribute of an element. For example, the following rule:

```
element "picture" {
   is fm graphic element;
   attribute "place" {
      is fm property position;
      value "left" is fm property value subcol left;
    }
   attribute "just" {
      is fm property alignment;
      value "left" is fm property value align left;
}}
```

  corresponds to an erroneous markup ATTLIST such as:

```
<!ATTLIST picture
   place (left, sright, snear, . . .)
   just (left, aright, acenter, . . .)
>
```

- The default declarations for graphic elements include three attributes that have a name token group as the declared value: `position`, `align`, and `impby`. These attributes correspond to the FrameMaker properties `position`, `alignment`, and `import by reference or copy`.

By default, the defined values and corresponding property values for the `position` property are as follows:

Defined value	Property value
`inline`	`inline`
`top`	`top`
`below`	`below`
`bottom`	`bottom`
`sleft`	`subcol left`
`sright`	`subcol right`
`snear`	`subcol nearest`
`sfar`	`subcol farthest`
`sinside`	`subcol inside`
`soutside`	`subcol outside`
`tleft`	`textframe left`
`tright`	`textframe right`
`tnear`	`textframe nearest`
`tfar`	`textframe farthest`
`tinside`	`textframe inside`
`toutside`	`textframe outside`
`runin`	`run into paragraph`

By default, the defined values and corresponding property values for the `alignment` property are as follows:

Defined value	Property value
`aleft`	`align left`
`aright`	`align right`
`acenter`	`align center`
`ainside`	`align inside`
`aoutside`	`align outside`

By default, the defined values and corresponding property values for the `import by reference or copy` property are as follows:

Defined value	Property value
`ref`	`reference`
`copy`	`copy`

- FrameMaker defines the `table border ruling` property for working with tables and the `alignment` and `vertical alignment` properties for working with colspecs and spanspecs.

  If you use the CALS table model for your tables, you should use read/write rules to translate these properties to the `frame`, `align`, and `valign` attributes on appropriate elements. There is also a default correspondence between the FrameMaker property values and the defined value in markup.

  If you do not use the CALS table model, you may still choose to translate these FrameMaker formatting properties to markup attributes. In this case, you must also determine the translation from property value to defined value.

- If you use the CALS table model, the `frame` attribute has the following defined values: `all`, `top`, `bottom`, `topbot`, `sides`, and `none`. The values for the corresponding `table border ruling` property are the same as the defined values, except that the `topbot` defined value is the `top and bottom` property value.

  The `align` attribute and the corresponding `cell alignment type` property have the following values: `left`, `center`, `right`, `justify`, and `char`.

  The `valign` attribute and the corresponding `vertical alignment` property have the following values: `top`, `middle`, and `bottom`.

Examples

- To use the `table border ruling` property for a non-CALS table and to set its name tokens, use this rule:

```
element "tab" {
   is fm table element;
   attribute "frame" {
      is fm property table border ruling;
      value "all" is fm property value all;
      value "top" is fm property value top;
      value "bottom" is fm property value bottom;
      value "topbot" is fm property value top and bottom;
      value "sides" is fm property value sides;
      value "none" is fm property value none;
}}
```

- To rename the FrameMaker `import by reference or copy` property as the `refcopy` attribute, and to also change the name tokens, use this rule:

```
attribute "refcopy" {
   is fm property import by reference or copy;
   value "r" is fm property value reference;
   value "c" is fm property value copy;
}
```

See also

Related rules	"fm property" on page 399 "is fm property" on page 424
Rules mentioned in synopses	"attribute" on page 367 "element" on page 376 "value" on page 464

is fm reference element

For SGML, use the `is fm reference element` rule to translate an entity in markup to an element defined on a reference page in a FrameMaker document (a reference element). For XML, use this rule to translate an internal entity to a reference element. Read/write rules do not support double-byte characters, so you cannot use this rule to process elements with double-byte characters in their names.

Synopsis and contexts

```
1. entity "ename" is fm reference element ["fmtag"];
2. reader entity "ename" is fm reference element ["fmtag"];
```

Arguments

ename	An entity name.
fmtag	A FrameMaker element tag.

Details

- For SGML, instead of using this rule to translate an SDATA entity, you can use a parameter literal of a particular form. For information on how to do so, see "Translating SDATA entities as FrameMaker reference elements" on page 263.
- For XML, SDATA entities are not allowed—this rule translates internal entities.
- You can use the `is fm reference element` rule within an `entity` rule at the highest level to have the translation occur in both directions. Or you can put the `entity` rule inside a `reader` rule to have the translation occur only when reading an SGML document into FrameMaker. Remember that the SDATA entity must be declared in the DTD in order to use this rule.
- The FrameMaker element must occur in a flow named `Reference Elements`. That flow must be on a reference page of the application's template file with a name that starts with `SGML Utilities Page`—for example, `SGML Utilities Page 1` or `SGML Utilities Page Logos`. For information on working with reference pages, see the FrameMaker user's manual.
- When FrameMaker encounters references to the specified entity while translating an markup document to FrameMaker, it copies the appropriate element from its reference

page in the FrameMaker template associated with the structure application. When it encounters an instance of an element associated with one of the reference pages while writing a FrameMaker document to markup, it generates an entity reference.

- When you use this rule, the *fmtag* element must be defined for your FrameMaker documents and valid in the contexts in which *ename* occurs. If it is not, the resulting FrameMaker document is invalid.

Examples

- Assume you have an entity named `legalese` which contains text you need to include in many places. The entity is too long to be a FrameMaker variable, and you don't want to treat it as an entire paragraph. Instead, you can choose to have the entity correspond to a text range element called `LegaleseFragment`. To do so, add the following rule to your rules document:

  ```
  entity "legalese" is fm reference element "LegaleseFragment";
  ```

  The entity declaration in your DTD looks like this for XML:

  ```
  <!ENTITY legalese "">
  ```

  The entity declaration in your DTD looks like this for SGML:

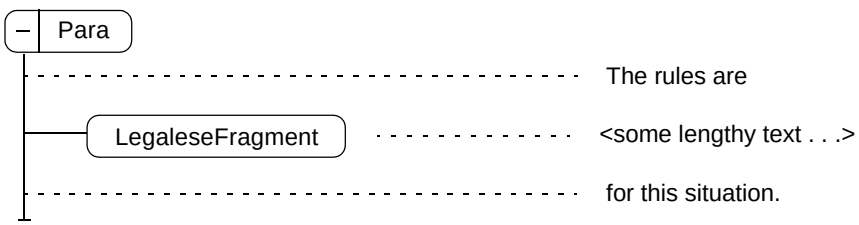
  ```
  <!ENTITY legalese SDATA "[]">
  ```

  Create a reference frame on the reference page of your application which contains the element "LegaleseFragment" with your boilerplate text. In order for the element to be treated as a "text range" use the appropriate TextFormatRules for this element in the EDD.

  When FrameMaker translates a markup document that contains the following markup:

  ```
  <para>The rules are &legalese; for this situation.</para>
  ```

  It produces the following element structure:



See also

Rules mentioned in synopses — "entity" on page 381

General information on this topic — Chapter 17, "Translating Entities and Processing Instructions"

is fm rubi element

Use the `is fm rubi element` rule to identify a markup element that translates to a Rubi element in FrameMaker. You can choose either to have the same name in both representations or to change the name during translation. Read/write rules do not support double-byte characters, so you cannot use this rule to process elements with double-byte characters in their names.

Synopsis and contexts

```
element "gi" {. . .
   is fm rubi element ["fmtag"];
. . .}
```

Arguments

gi	A generic identifier.
fmtag	A FrameMaker element tag.

Details

- If you use this rule, the other subrules of the `element` rule that you can use for the same markup element are as follows:

  `attribute`, page 367, specifies what to do with a markup element's attributes.

  `fm attribute`, page 396, specifies what to do with attributes present in the FrameMaker representation of the element but not in the markup representation.

Examples

- To translate the markup element `rubitext` to the `Rubitext` element in FrameMaker, use this rule:

  ```
  element "rubitext" is fm rubi group element;
  ```

- To translate it to the `MyRubiTextp` element, use this rule:

  ```
  element "rubitext" is fm rubi element "MyRubiText";
  ```

See also

Rules mentioned in synopses	"element" on page 376
General information on this topic	Chapter 16, "Translating Elements and Their Attributes"

is fm rubi group element

Use the `is fm rubi group element` rule to identify a markup element that translates to a Rubi group element in FrameMaker. You can choose either to have the same name in both representations or to change the name during translation. Read/write rules do not

support double-byte characters, so you cannot use this rule to process elements with double-byte characters in their names.

Synopsis and contexts

```
element "gi" {. . .
   is fm rubi group element ["fmtag"];
. . .}
```

Arguments

gi	A generic identifier.
fmtag	A FrameMaker element tag.

Details

- If you use this rule, the other subrules of the `element` rule that you can use for the same markup element are as follows:

  `attribute`, page 367, specifies what to do with a markup element's attributes.

  `fm attribute`, page 396, specifies what to do with attributes present in the FrameMaker representation of the element but not in the markup representation.

Examples

- To translate the markup element `rubigroup` to the `Rubigroup` element in FrameMaker, use this rule:

  ```
  element "rubigroup" is fm rubi group element;
  ```

- To translate it to the `MyRubiGroup` element, use this rule:

  ```
  element "rubigroup" is fm rubi group element "MyRubiGroup";
  ```

See also

Rules mentioned in synopses	"element" on page 376
General information on this topic	Chapter 16, "Translating Elements and Their Attributes"

is fm system variable element

Use the `is fm system variable element` rule to identify a markup element that translates to a system variable element in FrameMaker. You can choose either to have the same name in both representations or to change the name during translation. Read/write rules do not support double-byte characters, so you cannot use this rule to process elements with double-byte characters in their names.

Synopsis and contexts

```
element "gi" {. . .
   is fm system variable element ["fmtag"];
. . .}
```

Arguments

gi	A generic identifier.
fmtag	A FrameMaker element tag.

Details

- If you use this rule, the other subrules of the `element` rule that you can use for the same markup element are:

  `attribute`, page 367, specifies what to do with a markup element's attributes.

  `fm attribute`, page 396, specifies what to do with attributes present in the FrameMaker representation of the element but not in the markup representation.

- This rule does not apply to translating non-element FrameMaker variables.

Examples

- To translate the markup element `date` to the `Date` system variable element in FrameMaker, use this rule:

  ```
  element "date" is fm system variable element;
  ```

  You specify which system variable to use by adding a rule to the `Date` element's definition in the FrameMaker EDD. For example:

  Element (System Variable): Date

  System variable format rule

  In all contexts.
  Use system variable: Current Date (Long)

See also

Related rules	"is fm variable" on page 441 "fm variable" on page 402
Rules mentioned in synopses	"element" on page 376
General information on this topic	Chapter 21, "Translating Variables and System Variable Elements"

is fm table element

Use the `is fm table element` rule to identify a markup element that translates to a table element in FrameMaker. You can choose either to have the same name in both representations or to change the name during translation. Read/write rules do not support double-byte characters, so you cannot use this rule to process elements with double-byte characters in their names.

Synopsis and contexts

```
element "gi" {. . .
   is fm table element ["fmtag"];
. . .}
```

Arguments

gi	A generic identifier.
fmtag	A FrameMaker element tag.

Details

- If you use the CALS table model, you do not need to use this rule to translate the CALS `table` element properly.
- If your markup element declarations for a table element do not include an attribute that corresponds to the `columns` property, you must use the `fm property` rule to specify a number of columns for the table.
- If you use this rule, the other subrules of the `element` rule that you can use for the same markup element are as follows:

  `attribute`, page 367, specifies what to do with a markup element's attributes.

  `fm attribute`, page 396, specifies what to do with attributes present in the FrameMaker representation of the element but not in the markup representation.

  `fm property`, page 399, specifies what to do with FrameMaker properties associated with the element.

  `reader insert table` *`part`* `element`, page 410, indicates that the software should insert the indicated table part (table title, table heading, or table footing), even if the markup element structure or instance does not contain the corresponding element.

Examples

- To translate the markup element `gloss` to the `Gloss` table element in FrameMaker, use this rule:

  ```
  element "gloss" is fm table element;
  ```

- To translate it to the `Glossary` table element, use this rule:

  ```
  element "gloss" is fm table element "Glossary";
  ```

See also

Rules mentioned in synopses	“element” on page 376
General information on this topic	Chapter 18, “Translating Tables”

is fm table part element

Use the `is fm table` *`part`* `element` rule to identify a markup element that translates to a table part element in FrameMaker, such as a table title element. You can choose either to have the same name in both representations or to change the name during translation. Read/write rules do not support double-byte characters, so you cannot use this rule to process elements with double-byte characters in their names.

Synopsis and contexts

```
element "gi" {. . .
   is fm table part element ["fmtag"];
. . .}
```

Arguments

gi	A generic identifier.
part	A FrameMaker table part. One of the keywords: `title`, `body`, `heading`, `footing`, `row`, `cell`.
fmtag	A FrameMaker element tag.

Details

- If you use the CALS table model, you do not need to use this rule to translate elements representing parts of tables in CALS properly.
- If you map a markup element to a FrameMaker table part element, then the element cannot be used anywhere in the instance except as that table part. For example, if you have a “`title`” element and you use the following rule:

  ```
  element “title” is fm table title element;”
  ```

  Then you would not be able to insert a “`title`” element in a `Chapter` element.
- If you use this rule, the other subrules of the `element` rule that you can use for the same markup element are as follows:

  `attribute`, page 367, specifies what to do with a markup element’s attributes.

  `fm attribute`, page 396, specifies what to do with attributes present in the FrameMaker representation of the element but not in the markup representation.

  `fm property`, page 399, specifies what to do with FrameMaker properties associated with the element.

`reader end vertical straddle`, page 379, indicates that the associated table row or cell element terminates a vertical table straddle. This subrule applies only if *`part`* is `row` or `cell`.

`reader start new row`, page 458, indicates that the associated table cell element indicates the start of a new row in the table. This subrule applies only if *`part`* is `cell`.

`reader start vertical straddle`, page 460, indicates that the associated table cell element starts a vertical table straddle. This subrule applies only if *`part`* is `cell`.

Examples

- To translate the markup element `head` as the FrameMaker table heading element `Head`, use this rule:

```
element "head" is fm table heading element;
```

- To translate the markup element `dfn` as the FrameMaker table cell element `Definition`, use this rule:

```
element "dfn" is fm table cell element;
```

See also

Rules mentioned in synopses — "element" on page 376

General information on this topic — Chapter 18, "Translating Tables"

is fm text inset

Use the `is fm text inset` rule to translate a declared entity to a text inset in FrameMaker. While you can translate any entity to a text inset, we suggest you only do this with SDATA entities when working with SGML. Read/write rules do not support double-byte characters, so you cannot use this rule to process elements with double-byte characters in their names.

> **XML:** The XML standard does not allow SDATA entities, so you cannot use this rule for that purpose. FrameMaker translates external text entities as text insets by default, so this rule is not necessary for that type of entity.

Synopsis and contexts

```
1. entity "ename" is fm text inset "fname"
      [in body_or_ref flow "flowname"];

2. reader entity "ename" is fm text inset "fname"
      [in body_or_ref flow "flowname"];
```

Arguments

ename	An entity name.

fname	A filename containing the text to include. This file must be a FrameMaker document or a file of a type for which FrameMaker has a filter, for example, a MS-Word document.
body_or_ref	One of the keywords: `body` or `reference`, indicating the type of text flow in which to find the text to include. You can specify this option only if *fname* is a FrameMaker document.
flowname	The name of the FrameMaker text flow.

Details

- By default, external text entities in markup are imported as text insets. For the markup to be valid, the external text entities must be text, XML, or SGML files. In the FrameMaker document, the text insets use these files as their sources. It is probably most advantageous to retain these files for the text insets; you do not need to use the `is fm text inset` rule to import external text entities as text insets.
- The source file for the text inset must either be a FrameMaker file or a file of a format FrameMaker can filter automatically. You cannot use an SGML file as the source of the text inset.
- Instead of using this rule to translate an SGML SDATA entity to a text inset, you can use a parameter literal of a particular form. For information on how to do so, see "Translating SDATA entities as FrameMaker text insets" on page 262.
- You can use the `is fm text inset` rule within an `entity` rule at the highest level to have the translation occur in both directions. Or you can put the `entity` rule inside a `reader` rule to have the translation occur only when reading an SGML document into FrameMaker.
- If *fname* is not a FrameMaker document, you cannot specify the `in body flow` or `in reference flow` options. In this case, FrameMaker uses all of the text in the file specified by *fname* for the text inset.

  If *fname* is a FrameMaker document and you do not specify a flow to use, FrameMaker use the contents of the main body flow of the specified document.
- **Important:** *flowname* must exactly match the name of a flow in the document. If there is no match for the type of flow you specify (body or reference), then a crash will result. If there is more than one matching flow, FrameMaker uses the first matching flow.
- By default, the software reformats the text inset to conform to the format rules of the document containing the text inset. If the source for the text inset has element structure, FrameMaker also retains that element structure.

  You can confirm this behavior with the `reformat using target document catalogs` rule. You can change this behavior using the subrules `reformat as plain text` or `retain source document formatting`.

- FrameMaker requires that a structured flow have exactly one highest-level element. For this reason, you cannot use a single text inset to include multiple elements at the top level of the inset. You must use multiple text insets for this purpose.
- FrameMaker puts an end-of-paragraph symbol after a text inset. For this reason, you cannot use a text inset to insert a range of text inside a single paragraph. To do so, you can translate the entity either as a FrameMaker variable (with the `is fm variable` rule) or as a reference element (with the `is fm reference element` rule).

Examples

- Assume you have declared an SGML SDATA entity. You also have a single paragraph of boilerplate text to be used in your documents. You can place this text on a reference page in a text column with a flow called `BoilerPlate` in the FrameMaker template for your SGML application. If that template is the file `template.doc`, you could use this rule to translate occurrences of the `boiler` entity to a text inset in corresponding FrameMaker documents:

```
entity "boiler"
   is fm text inset "template.doc"
      in reference flow "BoilerPlate";
```

See also

Related rules	"reformat as plain text" on page 454 "reformat using target document catalogs" on page 455 "retain source document formatting" on page 456 "is fm reference element" on page 430 "is fm variable" on page 441
Rules mentioned in synopses	"entity" on page 381 "reader" on page 453
General information on this topic	Chapter 17, "Translating Entities and Processing Instructions"

is fm value

Use the `is fm value` rule to translate the value of a markup attribute to a particular choice for a FrameMaker choice attribute. The attribute's declared value must be a name token group or `NOTATION`.

Synopsis and contexts

```
1. value "token" is fm value "val";

2. attribute "attr" {. . .
      value "token" is fm value "val";
   . . .}
```

```
3. element "gi" {. . .
      attribute "attr" {. . .
         value "token" is fm value "val";
   . . .} . . .}
```

Arguments

token	A token in a name token group.
val	An allowed value for a FrameMaker choice attribute.
attr	The name of a markup attribute.
gi	A generic identifier.

Details

- Use this rule when the corresponding markup attribute translates to a choice attribute in FrameMaker. If the markup attribute translates to a FrameMaker property, you need to use the `is fm property value` rule to specify the correspondence between markup tokens and FrameMaker property values.

Examples

- If the token list `(r | b | g)` is used by multiple attributes, you can use these rules to translate the individual tokens consistently:

```
value "r" is fm value "Red";
value "b" is fm value "Blue";
value "g" is fm value "Green";
```

- If the token list `(r | b | g)` is used by several attributes as above but by the `bird` element differently, you can add this rule to the above rules:

```
element "bird" {is fm element;
   ] attribute "species" {
   value "r" is fm value "Robin";
   value "b" is fm value "Blue Jay";
   value "g" is fm value "Goldfinch";
}}]
```

See also

Related rules	“is fm property value” on page 426
Rules mentioned in synopses	“attribute” on page 367 “element” on page 376 “value” on page 464
General information on this topic	Chapter 16, “Translating Elements and Their Attributes”

is fm variable

Use the `is fm variable` rule to translate a declared markup text entity to a FrameMaker non-element variable.

Synopsis and contexts

```
1. entity "ename" is fm variable ["var"];
2. reader entity "ename" is fm variable ["var"];
```

Arguments

ename	An entity name.
var	A FrameMaker variable name.

Details

- You can use the `is fm variable` rule within an `entity` rule at the highest level to have the translation occur in both directions. Or you can put the `entity` rule inside a `reader` rule to have the translation occur only when reading a markup document into FrameMaker.

Examples

- To translate the markup element v to a non-element FrameMaker variable of the same name:

```
entity "v" is fm variable;
```

- To translate the FrameMaker variable `Licensor` to the markup element `lic`, use this rule:

```
entity "lic" is fm variable "Licensor";
```

See also

Related rules	“fm variable” on page 402 “is fm system variable element” on page 433
Rules mentioned in synopses	“entity” on page 381
General information on this topic	Chapter 21, “Translating Variables and System Variable Elements”

is processing instruction

On export, you use the `is processing instruction` rule to tell FrameMaker to create processing instructions for all non-element markers or for non-element markers of a particular type. By default, FrameMaker creates processing instructions for all non-element markers. You have the option of discarding non-element markers; you might use this rule

in conjunction with the drop rule when you want to discard some but not all non-element markers.

Synopsis and contexts

```
fm marker ["type_1", . . ., "type_n"] is processing instruction;
```

Arguments

type$_i$	A FrameMaker marker type, such as Index or Type 22.

Details

- If you do not supply any *type*$_i$ arguments, this rule applies to all non-element markers other than markers of the type reserved by FrameMaker for storing processing instructions, PI entities, and external data entities. (By default, the reserved marker types are `DOC PI` and `DOC Entity Reference`.)

Examples

- To discard all nonelement markers other than Index markers, use these rules:

```
fm marker "Index" is processing instruction;
fm marker drop;
```

See also

Rules mentioned in synopses	“fm marker” on page 398
General information on this topic	Chapter 22, “Translating Markers”

line break

Use the `line break` rule to tell FrameMaker about any limits on the length of lines in a markup file it generates. You also use it to tell the software whether or not to interpret line breaks in a markup document as FrameMaker paragraph breaks within elements.

Synopsis and contexts

```
1. reader line break is mode;

2. writer line break is mode;

3. element "gi" {. . .
      reader {. . .
         line break is mode;
   . . .} . . .}

4. element "gi" {. . .
      writer {. . .
         line break is mode;
   . . .} . . .}
```

Arguments

mode	For `writer`: *n* `characters` (where *n* is a positive integer in C syntax). For `reader`: one of `forced return` or `space`.
gi	A generic identifier.

Details

- This rule can be used at the highest level to set a default or within an `element` rule to set line breaks for only that element.
- On export, FrameMaker behaves as follows:

  When exporting the text of a paragraph, it ignores line breaks. It includes a space separating the two words on either side of a line break and attempts to avoid generating lines longer than *n* characters (the default is 80). It maintains a counter indicating how many characters it has placed on a single line. After this counter reaches *n*`-10`, it changes the next data character space to a record end.

  It generates a markup record end at the end of every paragraph and flow in the FrameMaker document.

  If you want a start-tag for an element and its contents to appear on the same line in the markup document, you must write aa structure API client.
- On import you have control over record ends not ignored by the underlying parser. Within a `reader` rule, *mode* can be one of the following:

  `forced return` informs FrameMaker that a line break within a text segment should be converted to a forced return.

space informs FrameMaker that a line break within a text segment should be treated as a space. This is the default.

Examples

- Line breaks may need to be treated differently within different elements. For example, a line break within an example element may need to be preserved on import, while a line break within a par element may be a word break:

```
element "example" reader line break is forced return;
element "par" reader line break is space;
```

marker text is

Use the marker text is rule to indicate whether the text of a marker element should become an attribute value or the content of the corresponding markup element. Note that the markup element must not be declared as empty if you want the marker text to be translated as content.

Synopsis and contexts

```
element "gi" {. . .
   is fm marker element ["fmtag"];
   marker text is attr_or_content;
. . .}
```

Arguments

gi	A generic identifier.
fmtag	A FrameMaker element tag.
attr_or_content	One of the keywords: attribute or content.

Details

- By default, FrameMaker translates a marker element in FrameMaker to a markup empty element. It writes the marker text as the value of the markup element's text attribute.
- Instead of the default, you can have FrameMaker translate a marker element to a markup element whose content model is #PCDATA. The marker text becomes the element's content.

Examples

- To state that the markup element mkr corresponds to the FrameMaker element Marker and to confirm the default behavior, you can use this rule:

```
element "mkr" {
   is fm marker element "Marker";
   marker text is attribute;
}
```

With this rule, the FrameMaker element definition:

Element (Marker): Marker

corresponds to the DTD declarations:

```
<!ELEMENT mkr - O EMPTY>
<!ATTLIST mkr text CDATA #IMPLIED
    type CDATA #IMPLIED>
```

In this case, if the FrameMaker document contains an instance of the `Marker` element whose marker text is “Some marker text” and whose type is Type 22, the markup output includes:

```
<mkr text="Some marker text" type="Type 22">
```

- To state that the markup element `mkr` corresponds to the FrameMaker element `Marker` but that the marker text should become element content in markup, you can use this rule:

```
element "mkr" {
   is fm marker element "Marker";
   marker text is content;
}
```

With this rule, the FrameMaker element definition:

Element (Marker): Marker

corresponds to the DTD declarations:

```
<!ELEMENT mkr (#PCDATA)>
<!ATTLIST mkr type CDATA #IMPLIED>
```

In this case, if the FrameMaker document contains an instance of the `Marker` element whose marker text is “Some marker text” and whose type is Type 22, the output includes:

```
<mkr type="Type 22">
Some marker text
</mkr>
```

See also

Rules mentioned in synopses	“element” on page 376 “is fm marker element” on page 423
General information on this topic	Chapter 22, “Translating Markers”

notation is

Use the `notation is` rule only in an `element` rule for a graphic or equation element, to provide information the software needs when writing a document containing graphics and equations to markup. FrameMaker uses this rule to determine the data content notation name to include in entity declarations it generates.

Synopsis and contexts

```
element "gi" {
   is fm graphic_or_eqn element ["fmtag"];
   writer type ["facetname"] notation is "notation";
. . .}}
```

Arguments

gi	A generic identifier.
graphic_or_eqn	One of the keywords: `graphic` or `equation`.
type	One of the rules `anchored frame`, `facet`, or `equation`. If `facet`, you must also supply the *facetname* argument. If *type* is `equation`, the rule applies to equation elements. If *type* is `facet`, the rule applies to a graphic element that contains only a single facet with the name specified by *facetname*. This occurs when the graphic element is an anchored frame containing only a single imported graphic object whose original file was in the *facetname* graphic format. You can use this rule with *type* set to `facet` multiple times if you want the software to treat several file formats differently. If *type* is `anchored frame`, the rule applies to a graphic element under all other circumstances.
facetname	A facet name. You supply this argument if and only if *type* is `facet`.
notation	A string representing a data content notation name.

Details

- By default, FrameMaker uses the first eight characters of the name of the facet it exports as the data content notation. If the graphic or equation has only internal FrameMaker facets, the software uses CGM as the data content notation.

Examples

- Assume your end users use the `af` graphic element within FrameMaker, creating the graphics using FrameMaker tools, but want to store them in TIFF format on export. Furthermore, you want to name the files based on the FrameMaker document's name, but with an extension of `.gr`. You can accomplish this with the following rule:

```
element "af" {
   is fm graphic element;
   writer anchored frame {
      notation is "TIFF";
      export to file "$(docname).gr";
}}
```

If you export the FrameMaker file `intro.doc`, the software writes the following entity declaration for the first instance of the `af` element that it finds:

```
<!ENTITY af1 SYSTEM "intro1.gr" NDATA TIFF>
```

See also

Related rules	"convert referenced graphics" on page 372 "entity name is" on page 383 "export to file" on page 389 "specify size in" on page 456
Rules mentioned in synopses	"element" on page 376 "is fm graphic element" on page 422 "is fm equation element" on page 420 "anchored frame" on page 365 "equation" on page 385 "facet" on page 394 "writer" on page 467
General information on this topic	Chapter 19, "Translating Graphics and Equations"

output book processing instructions

By default, when FrameMaker converts a FrameMaker book to markup, it puts `?FM book?` and `?FM document?` processing instructions in the markup document to indicate where the individual files in the FrameMaker documents began. You use the `output book processing instructions` rule to confirm or change this behavior.

Synopsis and contexts

```
writer [do not] output book processing instructions;
```

Arguments

None.

Details

- If you use the `generate book` rule to tell FrameMaker to use elements to identify book components when reading a markup document, you might choose to not have it output processing instructions when writing the book to markup. In this case, use this rule:

  ```
  writer do not output book processing instructions;
  ```

See also

Related rules	"generate book" on page 403

General information on this topic — Chapter 24, “Processing Multiple Files as Books”

preserve fm element definition

Use the `preserve fm element definition` rule to tell FrameMaker, when it is updating an EDD from a revised DTD, not to update the definition of a set of FrameMaker elements and their attributes on the basis of the DTD and other rules.

Synopsis and contexts

```
reader {. . .
   preserve fm element definition "fmtag1"[, . . ., "fmtagN"];
. . .}
```

Arguments

fmtag$_i$	A FrameMaker element tag.

Details

- FrameMaker uses the `preserve fm element definition` rule only when updating an EDD from a DTD. By default, when it updates an existing EDD, the software changes the definitions of FrameMaker elements to reflect the new DTD and all read/write rules. You may not want the definition of the FrameMaker element to change. For example, if one of your rules is to unwrap the element `body`, then any element with a definition that includes `body` will be modified directly include the contents of `body` instead of including `body`.

Examples

- Assume you have the rule:

  ```
  fm element "Body" unwrap;
  ```

  and the element definitions:

  Element (Container): Figure1
  General rule: Caption, Body

  Element (Container): Figure2
  General rule: Body, Footer

  Element (Container): Body
  General rule: Header, Line+

  The corresponding declarations are:

  ```
  <!ELEMENT figure1 - - (caption, header, line+)>
  <!ELEMENT figure2 - - (header, line+, footer)>
  ```

If you update the EDD containing the preceding definitions and use as input the DTD with the preceding declarations, FrameMaker replaces the definitions of `Figure1` and `Figure2` with:

Element (Container): Figure1
General rule: Caption, Header, Line+

Element (Container): Figure2
General rule: Header, Line+, Footer

If you wish to retain the original definitions of `Figure1` and `Figure2` in the revised EDD, include this rule:

```
reader preserve fm element definition "Figure1", "Figure2";
```

- Suppose you want to use aa structure API client to reverse the order of child elements in corresponding markup and FrameMaker elements. For example, assume you have the declaration:

  ```
  <!ELEMENT ex - - (a, b)>
  ```

  and the FrameMaker element definition:

  Element (Container): Ex
  General rule: B, A

  If you have no rules and update the EDD in this situation, FrameMaker updates the definition of `Ex` to correspond to the markup declaration. To suppress this change, use this rule:

  ```
  reader preserve fm element definition "Ex";
  ```

See also

Related rules "drop" on page 373
"unwrap" on page 461

preserve line breaks

Use the `preserve line breaks` rule to tell FrameMaker to keep line breaks for an element when importing and exporting markup documents. When importing markup, it translates every RE in the element as a forced return. When exporting markup, it translates forced returns as RE characters, and the line ends FrameMaker creates when automatically wrapping the text as non-RE line breaks in the markup file. This is useful for elements that use RE characters to insert white space in an element's content.

Synopsis and contexts

```
element {. . .
   preserve line breaks ;
. . .}
```

Arguments

None

Details

- For an element using this rule, the software writes a an RE (line break) immediately after the open tag and immediately before the close tag.
- For an element using this rule, on export, FrameMaker writes a space character entity reference and an RE (line break) for each necessary line break in the markup file. See the “line break” rule for information on how FrameMaker determines where to put these line breaks by default. Forced returns (shift-return) translate as RE characters (line breaks) in the markup file.
- For SGML, the space character entity uses the ISO entities reference (`&#SPACE`).
- For XML, no entity reference is written for the space character.
- For XML, this rule adds the `xml:space` attribute to the affected elements, with a value of `preserved`. This attribute directs XML applications to respect the whitespace characters in the element’s content. On import this attribute is retained—if the EDD for your template doesn’t specify an `xml:space` attribute for the given element, then that attribute will be invalid. You can either define this attribute in your EDD, or use read/write rules to drop the attribute on import.
- For export and import to have the same results, `preserve line breaks` must be specified for the same elements. For example, assume you use `preserve line breaks` on export for an element named `Code`. FrameMaker writes a space character entity reference and an RE (line break) when a line approaches the maximum line length, and it writes RE characters (line breaks) for forced returns. Now assume you remove preserve line breaks from the rules for the `Code` element. On import, FrameMaker will translate as spaces the space character entity reference/RE pairs, and as spaces any RE characters (line breaks) not removed by the parser (default behavior). Thus the forced returns (shift-return) are lost and the imported file is not the same as the exported file.
- When importing markup, `preserve line breaks` overrides the `line break is space` rule, if that rule is set. On import, `preserve line breaks` has the same effect for the specified element as the `line break is forced return` rule.

Examples

- The following rule preserves line breaks on import and export for the element named `code`:

```
fm element "code" {
   is fm element "Code";
   preserve line breaks;
}
```

See also

Rules mentioned in synopses: "element" on page 376

Related rules: "line break" on page 443

processing instruction

Use the `processing instruction` rule to drop processing instructions that are not recognized by FrameMaker. By default, the software stores such processing instructions as the marker text in non-element markers of type `DOC PI`.

Synopsis and contexts

```
processing instruction drop;
```

Arguments

None

Details

- When you translate a markup document to FrameMaker and the software encounters an unrecognized processing instruction such as:

  ```
  <?mypi?>
  ```

  it stores the processing instruction as the text of a non-element `DOC PI` marker by default, with the following as the marker text:

  ```
  mypi
  ```

  When you translate a FrameMaker document to markup, it outputs the corresponding processing instruction if it finds a non-element `DOC PI` marker with text in that format.

- This rule does not affect how FrameMaker treats the processing instructions it does recognize for books, book components, and other non-element markers.

Examples

- To discard all unrecognized processing instructions, use this rule:

  ```
  processing instruction drop;
  ```

See also

Rules mentioned in synopses: "drop" on page 373

General information on this topic: Chapter 17, "Translating Entities and Processing Instructions"

proportional width resolution is

Use the `proportional width resolution is` rule to change the number used as the total for proportional column widths in tables. By default, if FrameMaker writes proportional columns widths, those widths add to 100.

Synopsis and contexts

```
writer proportional width resolution is "value";
```

Arguments

value	An integer indicating the total for proportional column width values.

Details

- Using this rule does not indicate that FrameMaker uses proportional widths, only that if the software writes proportional widths, then those widths add to *value* instead of to 100. To tell FrameMaker to use proportional widths, you must include the `use proportional widths rule`.

Examples

- Assume you do not use the `proportional width resolution is` rule, but have this rule:

  ```
  writer use proportional widths;
  ```

  Further assume you have a 5-column table whose first two columns are 1 inch wide and whose last three columns are 2 inches wide. If the column widths are written to the `colwidth` attribute of the markup `table` element, then FrameMaker creates this start-tag for that table:

  ```
  <table colwidth="12.5* 12.5* 25* 25* 25*">
  ```

- Assume you have the same table as in the last example and you use this rule:

  ```
  writer {
     use proportional widths;
     proportional width resolution is "8";
  }
  ```

  FrameMaker writes this start-tag for the table:

  ```
  <table colwidth="1* 1* 2* 2* 2*">
  ```

- Assume you have the same table as in the previous examples and you use this rule:

  ```
  writer proportional width resolution is "8";
  ```

  That is, you do not also have the `use proportional widths` rule. In this case, FrameMaker ignores the “proportional width resolution” rule and writes this start-tag for the table:

  ```
  <table colwidth="1in 1in 2in 2in 2in">
  ```

See also

Related rules	“use proportional widths” on page 463
General information on this topic	Chapter 18, “Translating Tables”

put element

See “generate book” on page 403.

reader

The `reader` rule indicates a rule that applies only on import to FrameMaker. It can be used at the highest level to set a default, or within an `element` rule to specify information particular to that element.

Synopsis and contexts

```
1. element "gi" {. . .
      reader {. . .
         subrule;
   . . .} . . .}

2. reader {. . .
      subrule;
   . . .}
```

Arguments

gi	A generic identifier.
subrule	Valid subrules: `character map`, page 369, changes how FrameMaker translates between individual characters in the markup and FrameMaker character sets. Allowed only at the highest level. `drop content`, page 375, imports only the element itself, not its contents. Allowed only within an `element` rule. `end vertical straddle`, page 379, specifies the end of a vertical straddle in a table. Allowed only within an `element` rule for a table cell or row element. `entity`, page 381, specifies the treatment of an entity in FrameMaker. Allowed only at the highest level. `generate book`, page 403, specifies how to identify book components in a markup document. Allowed only at the highest level.

`insert table part element`, page 410, specifies that FrameMaker should generate a table part (table title, table heading, or table footing) even if there is no content for that part. Allowed only within an `element` rule for a table element.

`line break`, page 443, changes the treatment of line breaks in the markup instance which are not handled by the parser on import. Allowed at the highest level or within an `element` rule.

`preserve fm element definition`, page 448, instructs the software not to modify a FrameMaker element definition when updating an existing EDD. Allowed only at the highest level.

`start new row`, page 458, specifies that this table cell element starts a new row in the table. Allowed only within an `element` rule for a table row element.

`start vertical straddle`, page 460, specifies the start of a vertical straddle in a table. Allowed only within an `element` rule for a table cell element.

`table ruling style is`, page 461, specifies the ruling style to apply to all tables. Allowed only at the highest level.

Examples

- To change the default ruling style for tables:

```
reader table ruling style is "thick";
```

reformat as plain text

Use the `reformat as plain text` rule in an `entity` rule for an entity you want to translate as a text inset in FrameMaker. This specifies that the software should remove any element structure from the text inset and reformat the text using the format rules of the document into which the text inset is placed. You specify the other choices for formatting text insets with the rules `reformat using target document catalogs` and `retain source document formatting`.

Synopsis and contexts

```
1. entity "ename" {
      is fm text inset “fname”;
      reformat as plain text;
   . . .}

2. reader entity "ename" {
      is fm text inset “fname”;
      reformat as plain text;
   . . .}
```

Arguments

ename	An entity name.

See also

Related rules	“reformat using target document catalogs,” next “retain source document formatting” on page 456
Rules mentioned in synopses	“entity” on page 381 “is fm text inset” on page 437
General information on this topic	Chapter 17, “Translating Entities and Processing Instructions”

reformat using target document catalogs

Use the `reformat using target document catalogs` rule in an `entity` rule for an entity you want to translate as a text inset in FrameMaker. This specifies that the software should retain any element structure from the text inset and reformat the text using the format rules of the document into which the text inset is placed. This is the default behavior for entities treated as text insets. You specify the other choices for formatting text insets with the rules `reformat as plain text` and `retain source document formatting`.

Synopsis and contexts

```
1. entity "ename" {
      is fm text inset “fname”;
      reformat using target document catalogs;
   . . .}
2. reader entity "ename" {
      is fm text inset “fname”;
      reformat using target document catalogs;
   . . .}
```

Arguments

ename	An entity name.

See also

Related rules	“reformat as plain text,” (the previous section) “retain source document formatting” on page 456
Rules mentioned in synopses	“entity” on page 381 “is fm text inset” on page 437

General information on this topic	Chapter 17, “Translating Entities and Processing Instructions”

retain source document formatting

Use the `retain source document formatting` rule in an `entity` rule for an entity you want to translate as a text inset in FrameMaker. This specifies that the software should remove any element structure from the text inset, but keep the formatting of the source document, rather than reformatting it according to the rules of the document that contains the text inset. You specify the other choices for formatting text insets with the rules `reformat as plain text` and `reformat using target document catalogs`.

Synopsis and contexts

```
1. entity "ename" {
      is fm text inset “fname”;
      retain source document formatting;
   . . .}

2. reader entity "ename" {
      is fm text inset “fname”;
      retain source document formatting;
   . . .}
```

Arguments

ename	An entity name.

See also

Related rules	“reformat as plain text” on page 454 “reformat using target document catalogs,” (the previous section)
Rules mentioned in synopses	“entity” on page 381 “is fm text inset” on page 437
General information on this topic	Chapter 17, “Translating Entities and Processing Instructions”

specify size in

Use the `specify size in` rule only in an `element` rule for a graphic or equation element, to provide information the software needs when writing a document containing graphics and equations to markup. This rule determines which of the `dpi` or the `impsize` attribute FrameMaker uses to indicate the size of a graphic or equation. The rule also indicates what units are used for `impsize` and the resolution in which sizes are reported is always 0.001. If there is no `specify size in` rule, FrameMaker uses the `dpi` attribute.

Synopsis and contexts

```
element "gi" {
   is fm graphic_or_eqn element ["fmtag"];
   writer type ["facetname"] specify size in units
. . .}
```

Arguments

gi	A generic identifier.
graphic_or_eqn	One of the keywords: `graphic` or `equation`.
type	One of the rules `anchored frame`, `facet`, or `equation`. If `facet`, you must also supply the *facetname* argument. If *type* is `equation`, the rule applies to equation elements. If *type* is `facet`, the rule applies to a graphic element that contains only a single facet with the name specified by *facetname*. This occurs when the graphic element is an anchored frame containing only a single imported graphic object whose original file was in the *facetname* graphic format. You can use this rule with *type* set to `facet` multiple times if you want the software to treat several file formats differently. If *type* is `anchored frame`, the rule applies to a graphic element under all other circumstances.
facetname	A facet name. You supply this argument if and only if *type* is `facet`.
units	The units in which the size of the element is coded. Valid values: `cm`, `cc`, `dd`, `in`, `mm`, `pc`, `pi`, or `pt`.

Details

- Use this rule when you export FrameMaker documents to markup documents.
- FrameMaker reports the size of the elements in the indicated units, at a fixed resolution of .001.

Examples

- Suppose your document has a graphic element, `graph`, containing an Anchored Frame sized to fit a FrameMaker-drawn circle with a diameter of 3.15 centimeters. Given the rule:

```
element "graph" {
   is fm graphic element;
   writer anchored frame specify size in cm;
}
```

FrameMaker generates the attribute `height="3.150cm" and attribute width="3.150cm"`.

- However, with the same graphic, if the rule is:

```
element "graph" {
   is fm graphic element;
   writer anchored frame specify size in mm;
}
```

FrameMaker generates `height="31.500mm" and attribute width="31.500mm"`.

See also

Related rules	"convert referenced graphics" on page 372 "entity name is" on page 383 "export to file" on page 389 "specify size in" on page 456
Rules mentioned in synopses	"element" on page 376 "is fm graphic element" on page 422 "is fm equation element" on page 420 "anchored frame" on page 365 "equation" on page 385 "facet" on page 394 "writer" on page 467
General information on this topic	Chapter 19, "Translating Graphics and Equations"

start new row

Use the `start new row` rule in the `element` rule for a table cell element to specify that an occurrence of the table cell element indicates that FrameMaker should start a new table row to contain that cell.

Synopsis and contexts

```
element "gi" {. . .
   is fm table cell element ["fmtag"];
   reader start new row ["name"];
. . .}
```

Arguments

gi	A generic identifier.

fmtag	A FrameMaker element tag.
name	An optional name to identify this row

Details

- Your DTD may contain elements that you want to format as tables in FrameMaker even though the element hierarchy does not match that required by FrameMaker for tables. In such a situation, the nature of the element hierarchy may indicate where new rows should begin.
- In some cases, you can use a rule such as the following to indicate that a table cell starts a new row:

```
element "gi" {
   is fm table cell element;
   fm property column number value is "1";
}
```

  With this rule, when FrameMaker encounters a *gi* element, it tries to place that element in the first column of the current table row. If there is already a cell in the first column of the current row, the software automatically creates a new row for *gi*. In this situation, you would not also need the `start new row` rule.

  However, if there is not already a cell in the first column of the current row when the software encounters a *gi* element, it puts the *gi* cell in the current row and does not create a new row for it. This can happen if the table has a vertical straddle in the first column. When FrameMaker encounters a *gi* element on a row that should have a vertical straddle in the first column, with only the rule above, the software puts the *gi* element in the same row (because that cell isn't occupied). To guarantee a new row starts with the occurrence of *gi* instead, you should use this rule:

```
element "gi" {
   is fm table cell element;
   fm property column number value is "1";
   reader start new row;
}
```

Examples

- For a complete example using the `start new row` rule, see "Omitting explicit representation of table parts" on page 285.

See also

Related rules	"start vertical straddle," next
Rules mentioned in synopses	"element" on page 376 "is fm table part element" on page 436 "reader" on page 453

General information on this topic — Chapter 18, “Translating Tables”

start vertical straddle

Use the `start vertical straddle` rule inside the `element` rule for a table cell to specify that an occurrence of the cell element indicates the start of a vertical straddle.

Synopsis and contexts

```
element "gi" {. . .
   is fm table cell element ["fmtag"];
   reader start vertical straddle "name";
. . .}
```

Arguments

gi	A generic identifier.
fmtag	A FrameMaker element tag.
name	A name associated with a table straddle. This name must occur in at least one corresponding `end vertical straddle` rule.

Details

- Your DTD may contain elements that you want to format as tables in FrameMaker even though the element hierarchy does not match that required by FrameMaker for tables. In such a situation, the nature of the element hierarchy may indicate where vertical straddles should begin and end. The `start vertical straddle` rule allows you to specify such elements.
- Use this rule in conjunction with the `end vertical straddle` rule. That rule specifies a table cell or row that indicates the end of the vertical straddle started by this rule.
- You give a name to the particular straddle started by *gi*. In the corresponding `end vertical straddle` rule or rules, you use the same name to specify that the element ends this vertical straddle.

Examples

- For an example of the use of this rule, see “Creating parts of a table even when those parts have no content” on page 287.

See also

Related rules	“start new row,” (the previous section)
Rules mentioned in synopses	“element” on page 376 “is fm table part element” on page 436 “reader” on page 453

General information on this topic — Chapter 18, "Translating Tables"

table ruling style is

You use the `table ruling style is` rule to specify the ruling style for all tables.

Synopsis and contexts

```
reader table ruling style is "style";
```

Arguments

style	A ruling style for all tables. One of the keywords: None, Double, Medium, Thick, Thin, or Very Thin.

Details

- This rule specifies the ruling style applied to all tables. When working with the CALS table model, you can use the `frame`, `colsep`, and `rowsep` attributes to determine whether or not portions of a table have rulings. However, these attributes have boolean values. Consequently, you can only use them to say whether or not a table has a ruling, not what type of ruling to use if it does have one. In this situation, you could use the `table ruling style is` rule to set the ruling style for all tables.
- FrameMaker considers the ruling style set with this rule as custom ruling. If you re-import formats to the FrameMaker document and remove overrides, the ruling style set with this rule will remain. If possible, therefore, you should use table formats to specify ruling styles.

Examples

- To specify that all tables should use the Thick ruling style, use this rule:

```
reader table ruling style is "Thick";
```

See also

General information on this topic — Chapter 18, "Translating Tables"

unwrap

Use the `unwrap` rule when you do not want to preserve an element on translation from one representation to another. If you specify that FrameMaker should unwrap an element (*gi* or *fmtag*), the software places the element's content as part of the content of the element's parent element, but does not make an element for *gi* or *fmtag* itself.

Synopsis and contexts

```
1. element "gi" unwrap;
2. fm element "fmtag" unwrap;
```

Arguments

gi	A generic identifier.
fmtag	A FrameMaker element tag.

Details

- When FrameMaker encounters an element to be unwrapped, it does not insert a corresponding element into the document it is creating. Instead, it inserts the content of an unwrapped element.
- If you use this rule to unwrap FrameMaker cross-reference elements or system variable elements, those elements become text in the resulting markup document.
- When importing a DTD or exporting an EDD, FrameMaker does not generate an element definition or declaration corresponding to an element that is unwrapped. Furthermore, when an element uses the unwrapped element in its definition, the software replaces the name of the unwrapped element with its content model or general rule in the general rule or content model of the element that used it or replaces it with the list of its children in an exception. You can change this behavior by using the `preserve fm element definition` rule.
- You cannot use the `unwrap` rule with any other subrule of the `element` or `fm element` rules. For example, you cannot specify that a markup element both be unwrapped and be translated to a FrameMaker element.

Examples

- A markup document used to produce both the student's and teacher's edition of a textbook might include an `ANSWER` element used for answers to exercises. In producing the teacher's edition of the textbook, this element might be unwrapped into FrameMaker as text. A structure API client could associate this element with the condition tag `Answer`.
- Suppose a DTD contains the following declarations:

```
<!ELEMENT wrapper - - (a, b)>
<!ELEMENT x - - (p, q, wrapper, r)>
<!ELEMENT y - - (#PCDATA) +(wrapper)>
```

  and you have this rule:

```
element "wrapper" unwrap;
```

  FrameMaker would generate the following element definitions:

  Element (Container): X
  General rule: P, Q, A, B, R

  Element (Container): Y
  General rule: <TEXT>
  Inclusions: A, B

See also

Related rules	“preserve fm element definition” on page 448 “drop” on page 373
Rules mentioned in synopses	“element” on page 376 “fm element” on page 397
General information on this topic	Chapter 16, “Translating Elements and Their Attributes”

use processing instructions

See “generate book” on page 403.

use proportional widths

Use the `use proportional widths` rule to indicate that when FrameMaker writes the width of table columns, it should use proportional measurements. By default, if the software writes the width of table columns, it uses absolute measurements.

Synopsis and contexts

```
writer use proportional widths;
```

Arguments

None.

Details

- If you use this rule when writing an attribute indicating the width of one or more columns in a table, FrameMaker writes values such as `"25*"`, where the asterisk `*` indicates a proportional measurement, instead of values such as `"0.25in"` which are absolute measurements.
- If you use this rule, you can also use the `proportional width resolution is` rule to determine what number the values add to. Without the `proportional width resolution is` rule, the proportional measurements add to 100.

Examples

- Assume you do not use the `proportional width resolution is` rule, but have this rule:

  ```
  writer use proportional widths;
  ```

  Further assume you have a 5-column table whose first two columns are 1 inch wide and whose last three columns are 2 inches wide. If the column widths are written to the `colwidth` attribute of the markup `table` element, then FrameMaker creates this start-tag for that table:

  ```
  <table colwidth="12.5* 12.5* 25* 25* 25*">
  ```

- Assume you have the same table as in the last example and you use this rule:

```
writer {
   use proportional widths;
   proportional width resolution is "8";
}
```

FrameMaker writes this start-tag for the table:

```
<table colwidth="1* 1* 2* 2* 2*">
```

See also

Related rules	“proportional width resolution is” on page 452
General information on this topic	Chapter 18, “Translating Tables”

value

Use the `value` rule to translate the value of a markup attribute to the value of a particular FrameMaker property or to a particular choice for a FrameMaker choice attribute. The attribute’s declared value must be a name token group or `NOTATION` and a name token group.

Synopsis and contexts

1. **value** "*token*" *subrule*;

2. attribute "*attr*" {. . .
   value "*token*" *subrule*;
   . . .}

3. element "*gi*" {. . .
   attribute "*attr*" {. . .
   value "*token*" *subrule*;
   . . .} . . .}

Arguments

token	A token in a name token group.
attr	The name of a markup attribute.
gi	A generic identifier.
subrule	One of the following: `is fm value`, page 439, translates a markup value to a particular choice for a FrameMaker choice attribute. `is fm property value`, page 426, translates a markup value to the value of a particular FrameMaker property.

Details

- The rule can be used at the highest level to set a default, within a highest-level attribute rule to set the default for all attributes that use that token, or within an `element` rule to set the default for a particular token within a particular attribute in that element.

Examples

- To rename the FrameMaker `import by reference or copy` property as the `refcopy` attribute, and to also change the name tokens, use this rule:

```
attribute "refcopy" {
   is fm property import by reference or copy;
   value "r" is fm property value reference;
   value "c" is fm property value copy;
}
```

- If the token list `(r | b | g)` is used by multiple attributes, you can use these rules to translate the individual tokens consistently:

```
value "r" is fm value "Red";
value "b" is fm value "Blue";
value "g" is fm value "Green";
```

- If the token list `(r | b | g)` is used by several attributes as above, but by the `bird` element differently, you can add this rule to the above rules:

```
element "bird" {is fm element;
] attribute "species" {
   value "r" is fm value "Robin";
   value "b" is fm value "Blue Jay";
   value "g" is fm value "Goldfinch";
   }}]
```

See also

Related rules	“is fm value” on page 439
	“is fm element” on page 419
Rules mentioned in synopses	“attribute” on page 367
	“element” on page 376
General information on this topic	Chapter 16, “Translating Elements and Their Attributes”

value is

See “fm property” on page 399.

write structured document

By default, when you save a FrameMaker document to markup, the software writes out the document instance, any declarations for the internal DTD subset, and a DOCTYPE statement which references the external DTD subset, but (for SGML) not an SGML declaration nor the declarations within the external DTD subset. You can use this rule to confirm the default behavior.

Synopsis and contexts

```
writer write structured document;
```

Arguments

None.

Details

- You cannot use the `write structure document` rule and the `write sgml document instance only` rule in the same read/write rules file.

See also

Related rules

"external dtd" on page 392
"include dtd" on page 408
"include sgml declaration" on page 409
"write structured document instance only," next

write structured document instance only

By default, when you save a FrameMaker document to markup, the software writes out the document instance, any declarations for the internal DTD subset, and a DOCTYPE statement which references a file for the external DTD subset. For SGML, it doesn't write an SGML declaration. This rule causes the software to write the document instance only--no external or internal DTD and no SGML declarations.

Synopsis and contexts

```
writer write structured document instance only;
```

Arguments

None.

Details

- By default, when you translate a FrameMaker document to markup, as its last step the software runs the parser on the markup document to check its validity. If you use this rule, FrameMaker does not write a complete markup document and so does not send the result through the parser.

- You cannot use the `write structure document instance only` rule in the same read/write rules file as any of the `write structure document`, `include dtd`, or `include sgml declaration` rules.

See also

Related rules
"external dtd" on page 392
"include dtd" on page 408
"include sgml declaration" on page 409
"write structured document," (the previous section)

writer

The `writer` rule indicates a rule that applies only on export of a FrameMaker document to markup. It can be used at the highest level to set a default or within an `element` rule to specify a subrule for that element.

Synopsis and contexts

```
1. writer {. . .
      subrule;
   . . .}

2. element "gi" {. . .
      writer {. . .
         subrule;
   . . .} . . .}
```

Arguments

gi — A generic identifier.

subrule — Valid subrules:

`anchored frame`, page 365, tells FrameMaker what to do with graphic elements other than those with a single non-internal FrameMaker facet. Allowed only within an `element` rule for a graphic element.

`character map`, page 369, determines the correspondence between individual characters in the FrameMaker and markup character sets. Allowed only at the highest level.

`convert referenced graphics`, page 372, tells the software to create new files for graphic files that were imported by reference.`drop content`, page 375, exports a FrameMaker element without its contents. Allowed only within an `element` rule.

`equation`, page 385, tells FrameMaker what to do with equation elements. Allowed only with an `element` rule for an equation element.

`external dtd`, page 392, specifies an external DTD to use. Allowed only at the highest level.

`facet`, page 394, tells FrameMaker what to do with a graphic element that has a single non-internal FrameMaker facet. Allowed only with an `element` rule for a graphic element.

`[do not] include dtd`, page 408, specifies information to exclude or include in the written document. Allowed only at the highest level.

`[do not] include sgml declaration`, page 409, specifies information to exclude or include in the written document. Allowed only at the highest level.

`line break`, page 443, specifies treatment of line breaks not handled by the parser on export. Allowed at the highest level or within an `element` rule.

`[do not] output book processing instructions`, page 447, specifies whether or not to create processing instructions that identify book components when writing a FrameMaker book as a markup document. Allowed only at the highest level.

`proportional width resolution is`, page 452, specifies the total value to which proportional widths for table columns add up. Allowed only at the highest level.

`use proportional widths`, page 463, specifies that the software should use proportional values in describing the widths of table columns. Allowed only at the highest level.

`write sgml document`, page 466, specifies that an entire SGML document should be written, not just the document instance. This is the default. Note that the external DTD subset is not written to the file. Instead, a DOCTYPE statement with a reference to the external DTD file is written. Allowed only at the highest level.

`write sgml document instance only`, page 466, specifies that only the document instance should be written, not the DTD and SGML declaration. Allowed only at the highest level.

Examples

- To tell FrameMaker not to use processing instructions to identify book components when writing a FrameMaker book as a markup document, use this rule:

```
writer do not output book processing instructions;
```

- Assume you want all graphics to be exported in TIFF format. Further assume that some of your graphic elements were imported from the TIFF format. For these elements you don’t want to create a new external data entity. To accomplish this, use these rules:

```
element "graphic" {
   is fm graphic element;
 writer facet default{
 convert referenced graphics;
 export to file “$(entity) .tif as “TIFF”;
   writer anchored frame
      export to file "$(entity).tif" as "TIFF";
}
```

A Conversion Tables for Adding Structure to Documents

You can set up a conversion table to help end users automate the task of adding structure to documents. The conversion table uses paragraph and character formats to identify which document objects to wrap in elements, and element tags to identify which child elements to wrap in parent elements. A user wraps all of a document's contents in one move by applying a structure command to the document and referring to one of your conversion tables.

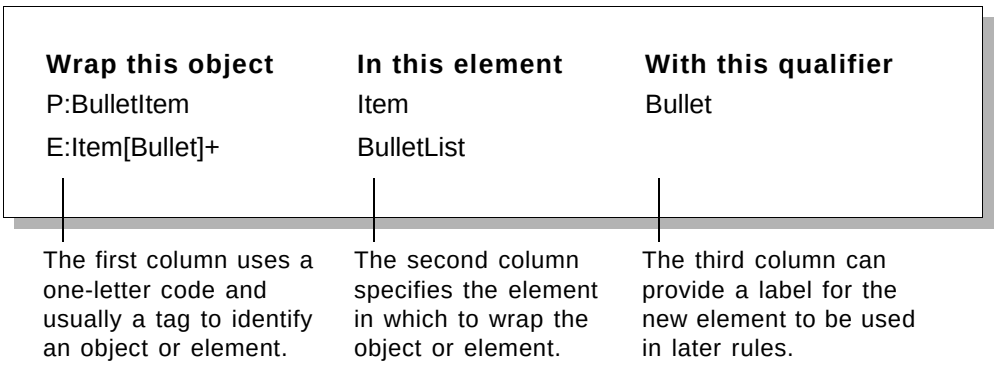
This appendix describes how to set up a conversion table and define object and element mapping in it. For information on the commands for adding structure to documents, see the FrameMaker user's manual

How a conversion table works

A conversion table contains rules for mapping between document objects and elements and between child elements and parent elements. The table is a regular FrameMaker table, with at least three columns and one body row. Each body row holds one rule.

The first column in a conversion table specifies a document object, a child element, or a sequence of child elements or paragraphs to wrap in an element. A *document object* is a paragraph, text range, table, table part (such as heading or row), equation, variable, footnote, Rubi group, Rubi text, marker, cross-reference, text inset, or graphic (anchored frame or imported graphic object).

The second column in the table specifies the element in which you want to wrap the object, child element, or sequence. The third column can specify an optional *qualifier* to use as a temporary label for the element in rules that are applied later. For example:



To add structure to a document or book, an end user chooses the Structure Current Document, Structure Documents, or Structure Current Book command from the File>Utilities submenu and refers to one of the conversion tables.

When someone adds structure to a document manually, he or she typically begins with the lowest-level components and works up to the highest level. For example, to add structure to a chapter an end user might start by wrapping sub-paragraph objects like text ranges and tables, then wrap the contents of paragraphs together in `Paragraph` elements, then wrap sequences of `Head` and `Paragraph` elements in `Section` elements, and so on until the entire document is wrapped in a single highest-level `Chapter` element.

The process of adding structure with a conversion table is similar to adding structure manually. FrameMaker begins by applying rules to document objects below the paragraph level, then applies rules at the paragraph level, and proceeds through successively higher levels. The process stops when FrameMaker reaches a single highest-level element or when no more rules can be applied. To understand this process, it helps to have manually structured a document.

Using the sample table above, FrameMaker first wraps each paragraph with the paragraph format `BulletItem` in an element called `Item` and gives the element a qualifier called `Bullet`. Then it wraps each `Item` element with the qualifier `Bullet` in a parent element called `BulletList`.

FrameMaker tries to order the rules as much as possible. If a rule needs a building block that is generated by a later rule, the later rule is run first so that all of the building blocks in the first rule are available. To make a conversion table easy to interpret for a human reader, you may want to write the rules in the order they should be applied.

Setting up a conversion table

You can have FrameMaker generate an initial conversion table for you from an unstructured document or book, or you can create a conversion table entirely from scratch. If you already have a document that end users need to add structure to, or a document that is similar to one users will add structure to, you'll probably want to let FrameMaker generate the initial table. You can modify the rules in the table as necessary.

After creating a conversion table, you can update it from other unstructured documents. Updating a table adds rules for any objects in the document that are not yet in the table.

A conversion table document can have the conversion table itself (which may be split up into several tables) and text or graphics you want to include for documenting the rules. It cannot have any tables other than conversion tables. You need to save the document before it can be used for adding structure to other documents or books.

Each body row in a conversion table holds one mapping rule. FrameMaker reads only the information in the first thee columns of the body rows, so you can use additional columns and headings and footings for comments about rules.

For information on defining and modifying the rules in a table, see "Adding or modifying rules in a conversion table" on page 474.

Generating an initial conversion table

You can have FrameMaker generate a conversion table from an unstructured document. This is the easiest way to begin a new conversion table.

To generate an initial conversion table, choose Generate Conversion Table from the File>Developer Tools submenu in a document with objects you want to structure. Select Generate New Conversion Table in the dialog box and click Generate.

The software looks through the flows on body pages in the document and compiles a list of every object that can be structured. For each object, it gives the object type and the format tag used in the document (if the object has a format), and maps the object to an element. The element tag is the same as the format tag, or if the object does not have a format, the element tag is a default name such as `CELL` or `BODY`. If necessary, FrameMaker removes parentheses and other characters to create an element tag that is valid.

The initial conversion table gives you a first pass through the document, identifying objects to wrap in elements. It does not identify child elements to wrap in parent elements—you need to add those rules to the table yourself.

This is an example of an initial conversion table:

Wrap this object	In this element	With this qualifier
P:Head1	Head1	
P:Head2	Head2	
P:Body	Body	
P:Code	Code	
SV:Current Date \(Long\)	CurrentDateLong	
C:Code	cCode	
TC:	CELL	
TR:	ROW	

For details on the object type identifiers used in the table (such as `P:` and `TC:`), see "Identifying a document object to wrap" on page 476.

Note that if there are conflicts in a format tag from the unstructured document, an object type identifier in lowercase is prepended to any duplicate element tag. In the example above, the element tag for text ranges with the `Code` character format is `cCode` because the document also has a paragraph format called `Code`.

When you create an initial table, FrameMaker does not examine the document's format catalogs—it looks only at objects actually used in the document. For this reason, the table may not be as complete as you need. You may want to update the table from a set of documents that together provide all or most of the objects you need rules for. You can also add and modify rules manually.

Setting up a conversion table from scratch

You can set up a regular FrameMaker table to serve as a conversion table. The table must appear on a body page in its own document. The document and table can be structured or unstructured. Begin a conversion table this way if you do not yet have an unstructured document to use for generating the table.

To set up a conversion table from scratch, create a new document and insert a table with at least three columns and one body row. The table can have any number of heading or footing rows.

You can divide a conversion table into several smaller tables. This is helpful when you have many rules and want to organize the rules in groups. Each table must have at least three columns and one body row. You can add explanatory heads and paragraphs between the tables to document the rules. Do not include tables that are not conversion tables.

Updating a conversion table

After creating a conversion table, you may want to update the table from at least one other unstructured document to get a more complete list of objects. FrameMaker adds a rule for each object from the document that is not already listed in the table.

To update a conversion table, choose Generate Conversion Table from the File>Developer Tools submenu in a document with the objects you want to structure. Select the name of the conversion table document in the Update Conversion Table popup menu and click Generate.

When you update a conversion table, the process that FrameMaker goes through is similar to the process of generating an initial table. The software does not examine the document's format catalogs—it looks only at objects actually used in the document.

Adding or modifying rules in a conversion table

Each body row in a conversion table holds one mapping rule. Follow these steps to define a mapping rule:

1. **In the first column, identify a document object, a child element, or a sequence of child elements or paragraphs to wrap.**

   You use a one- or two-letter code to identify the type of item and, in most cases, a format or element tag to narrow the definition. See "Identifying a document object to wrap" on page 476, "Identifying an element to wrap" on page 477, or "Identifying a sequence to wrap" on page 478.

2. **In the second column, specify an element in which to wrap the object, child element, or sequence.**

   Type one valid element tag. If you are writing rules for a document that already has element definitions, use tags from the document's Element Catalog.

   If you are wrapping a table part, graphic, or inset, FrameMaker always wraps all instances of the object in the same kind of element. The element has a default tag, such as `CELL`, `BODY`, `GRAPHIC`, or `INSET`. Type a different tag in the second column only if you want to override the default tag.

   You can also give an element an attribute with a value. For details, see "Providing an attribute for an element" on page 479.

3. **(Optional) In the third column, add a qualifier for the new element tag.**

   A qualifier is a temporary label that you can attach to an element tag for the structuring process. If you wrap the element in a parent element in a later rule, you include the qualifier tag with the element tag. For details, see "Using a qualifier with an element" on page 480.

To make a conversion table easy to read and to help you think through the process, we recommend that you put the rules in order from the lowest level to the highest. In the first rows of the table, write rules that wrap individual document objects such as text ranges, tables, and paragraphs; next add rules that wrap child elements in parent elements; then add rules that wrap sequences in elements; and finally add rules that wrap elements in one highest-level element.

Every flow in a document must have a highest-level element, and the element can be different for each flow.

About tags in a conversion table

Format and element tags in a conversion table are case-sensitive and must be specified the way they are defined in their catalogs. Qualifier tags are also case-sensitive, and two occurrences of one qualifier must match exactly. The following characters are not allowed in an element tag, but can appear in a format or qualifier tag if you precede them with a backslash (\) in the table:

```
( ) & | , * + ? % [ ] : \
```

A space character does not need to be preceded with a backslash. For example, you can write the tag `Format A`.

You can use a percentage sign (%) as a wildcard character in a format or element tag to match zero, one, or more characters. For example, `P:%Body` matches paragraphs with the format tag `Body`, `FirstBody`, or `BulletBody`.

Identifying a document object to wrap

To identify a document object to wrap in an element, type an object type identifier and (optionally) a format tag in the first column of the table. Separate the identifier and format tag with a colon.

FrameMaker finds all the objects with that type and format and wraps them in the element you specify in the second column of the table. If you leave the format tag out of the rule, FrameMaker finds all the objects with the specified type that are not identified in other conversion rules.

For example:

Wrap this object	In this element
P:Body	Para
T:RulesTbl	RulesTbl
T:	StandardTbl
Q:Small	SmallEqns

This rule wraps all tables not named in other rules, regardless of format tag.

These are the object type identifiers and format tags you can use:

Object type	Identifier	Format tag
Paragraph	`P:`	Paragraph format tag
Text range	`C:`	Character format tag
Table	`T:`	Table format tag
Table title	`TT:`	(none)
Table heading	`TH:`	(none)
Table body	`TB:`	(none)
Table row	`TR:`	(none)
Table cell	`TC:`	(none)
System variable	`SV:`	Variable format name
User variable	`UV:`	Variable format name
Graphic (anchored frame or imported object)	`G:`	(none)
Footnote	`F:`	Location of footnote: `Table` or `Flow`
Rubi group	`RG:`	(none)
Rubi text	`R:`	(none)
Marker	`M:`	Marker type

Object type	Identifier	Format tag
Cross-reference	X:	Cross-reference format name
Text Inset	TI:	(none)
Equation	Q:	Size of equation: Small, Medium, or Large

Table parts, graphics, and text insets do not have any formatting information, so FrameMaker wraps all instances of those objects in the same kind of element. The element has a default tag, such as CELL, BODY, GRAPHIC, or INSET. (Specify a different tag in the second column to override the default tag.)

You can write identifiers and the keywords for footnote location or equation size in any combination of uppercase and lowercase letters. The names of formats and marker types are case-sensitive, however, and must be typed the way they are specified in their catalogs.

A system variable can be wrapped in a variable element but a user variable cannot. If you identify a user variable, FrameMaker wraps it in a container element with the tag specified in the second column.

FrameMaker wraps a text inset in a container.

Identifying an element to wrap

To identify a child element to wrap in a parent element, type the object type identifier E: followed by an element tag and (optionally) a qualifier in brackets in the first column of the table. The qualifier must already be defined for the element in a rule applied earlier.

FrameMaker finds all instances of the element and wraps them in the element you specify in the second column of the table. You can omit the element tag if you include a qualifier.

For example:

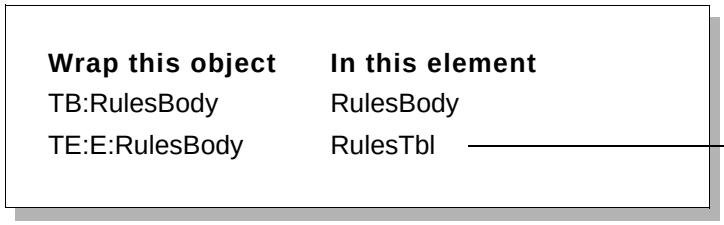
Wrap this object	In this element
E:Item[Bullet]	BulletItem
E:[1Head]	ChapHead

This rule wraps all elements with the qualifier 1Head not named in other rules.

You can type the E: identifier in either uppercase or lowercase. The element tags are case-sensitive, however, and must be typed the way they are specified in their catalog. You can even omit the E: identifier—when FrameMaker reads an object name with no identifier, it assumes the object is an element.

To identify a table child element to wrap in a table parent element, type the object identifier TE: followed by E:, an element tag, and (optionally) a qualifier in brackets in the first column of the table. This allows you to name a table element from one or more child elements, rather than naming it from a table format tag (with the T: identifier).

For example:



Most often, you wrap multiple elements together in one parent. You can use `E:` or `TE:` to identify a sequence of elements for this. For more information, see "Identifying a sequence to wrap," next. For more information on qualifiers, see "Using a qualifier with an element" on page 480.

Identifying a sequence to wrap

You can wrap a sequence of child elements in a parent element. For example, you might wrap a `Head` element followed by one or more `Paragraph` and `List` elements in a higher-level `Section`.

You can also wrap a sequence of unwrapped paragraphs in an element. For example, you might wrap a sequence of paragraphs with the format tag `Body` all in one `Note` element. (With other unwrapped document objects such as tables, graphics, and text ranges, you can wrap only one object in an element.)

To identify a sequence to wrap, specify object type identifiers and element tags or paragraph format tags, and use symbols to further describe the sequence. You can mix elements and unwrapped paragraphs together in one specification.

These are the symbols you can use:

Symbol	Meaning
Plus sign (+)	Item is required and can occur more than once.
Question mark (?)	Item is optional and can occur once.
Asterisk (*)	(SGML only) Item is optional and can occur more than once.
Comma (,)	Items must occur in the order given.
Ampersand (&)	Items can occur in any order.
Vertical bar (|)	Any one of the items in the sequence can occur.
Parentheses	Beginning and end of a sequence.

The symbols available are the same connectors, occurrence indicators, and parentheses used in general rules in an EDD. For more information on the symbols, see "Writing an EDD general rule" on page 111.

For example:

To identify this sequence	Use this specification
One or more `Item` elements	`Item+`
An element tagged `Item[Bullet]` followed by one or more unwrapped paragraphs tagged `Bullet`	`E:Item[Bullet], P:Bullet+`
A `ChapNum` element followed by a `ChapName` element	`ChapNum, ChapName`
A `Head` element followed by zero or more `Paragraph`, `BulletList`, or `NumberList` elements	`Head, (Paragraph|BulletList|NumberList)*`
An `Item[FirstNItem]` element followed by one or more `Item[NItem]` elements	`Item[FirstNItem], (Item[NItem])+` or `[FirstNItem], ([NItem])+`
A `RulesTitle` table title element followed by a `RulesBody` table body element	`TE:E:RulesTitle, E:RulesBody`

Providing an attribute for an element

When you specify an element in the second column of the table, you can provide an attribute for the element. In the structured document, all the element instances will have the attribute name and value.

To provide an attribute for an element, type the attribute name and value in brackets after the element tag in the second column of the table. Separate the name and value with an equal sign, and enclose the value in double quotation marks.

For example:

Wrap this object	In this element
P:Intro	Para[Security="Unclassified"]
P:Important	Note[Label="Important"]
E:Item+	List[Type="Numbered"]

If the unstructured document has an Element Catalog with an element and attribute matching the one you're providing, the attribute is the type specified in the catalog. If the attribute does not match an attribute already defined, the type is string.

If you need to use a double quotation mark in an attribute value, escape the quotation mark with a backslash (\). Other restrictions on characters are determined by the attribute's type.

(The string type allows any arbitrary text string.) For information on these restrictions, see "Attribute type" on page 172.

To give an element more than one attribute, separate the attribute definitions with an ampersand (&). For example, this specification gives the element a `Type` attribute with the value `Numbered` and a `Content` attribute with the value `Procedure`:

```
List [Type="Numbered" & Content="Procedure"]
```

For an example of an attribute that maintains formatting information from a qualifier, see "Using a qualifier with an element," next.

Using a qualifier with an element

Qualifiers act as temporary labels that preserve formatting information from the unstructured document until all elements have been wrapped. Qualifiers are used only in the conversion table—they do not show up in a final structured document.

To use a qualifier with an element specified in the second column of the table, type the qualifier tag in the third column. Then when you wrap the element in a later rule, type the qualifier tag in brackets after the element tag in the first column. Spell and capitalize the qualifier the same way in the two places. FrameMaker keeps track of qualifiers separately from elements, so you can use the same tag for an element and its qualifier.

For example:

Wrap this object	In this element	With this qualifier
P:BulletItem	Item	bulleted
P:NumberItem	Item	numbered
E:Item[bulleted]+	BulletList	
E:Item[numbered]+	NumberList	

First specify the qualifier for the element.

Then include the qualifier with the element in later rules.

In the example above, an unstructured document has both bulleted items and numbered items, with paragraph formats called `BulletItem` and `NumberItem`. When adding structure to the document, you want to wrap all the items in an `Item` element with a parent element of either `BulletList` or `NumberList`. To do this, you need to keep the `BulletItem` and `NumberItem` formatting designations long enough to determine in which list to wrap the items. The conversion table first associates qualifiers called `bulleted` and `numbered` with new `Item` elements. Then it wraps each `Item` element in either a `BulletList` or a `NumberList`, as specified by its qualifier.

Note that if you specify an attribute for formatting information in the second column, you cannot use the attribute as a label for preserving formatting during the conversion process. You still need to use the qualifier. For example:

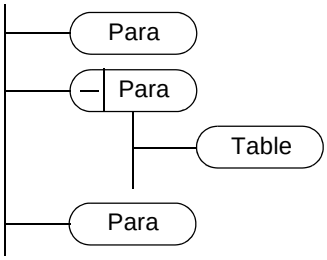
Wrap this object	In this element	With this qualifier
P:BulletItem	Item	bulleted
P:NumberItem	Item	numbered
E:Item[bulleted]+	List[Type="Bulleted"]	
E:Item[numbered]+	List[Type="Numbered"]	

Handling special cases

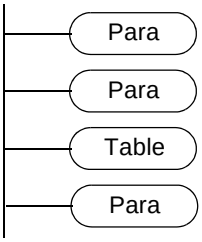
You may need to accommodate a few special circumstances or requirements in a conversion table.

Promoting an anchored object

In an unstructured FrameMaker document, a table or an anchored graphic must be anchored in a paragraph. The anchor specifies which paragraph to keep the object with as an author continues to edit the document. When a user adds structure to the document, the table or graphic normally becomes a child of the paragraph with the anchor, like this:



In a structured document, you often want a table or graphic element to be at the same level as its surrounding paragraph elements. FrameMaker can break the table or graphic out of its paragraph and promote the element to be a sibling of the paragraphs, like this:



To break a table or graphic out of its paragraph and promote it one level, add the keyword `promote` in parentheses after the element tag for the table or graphic. (The keyword is not case-sensitive.) For example:

Wrap this object	In this element
T:Table	Table (promote)

Note that FrameMaker promotes the object at the location of the anchor symbol in the paragraph. If the symbol is in the middle of the paragraph, the structured document will have half of the paragraph, then the table, and then the other half of the paragraph. Typically, you want the symbol to be at the end of the paragraph.

Flagging format overrides

An unstructured document may have *format overrides*. This happens when someone uses the Paragraph or Character Designer to make formatting changes to a paragraph or text range but does not save the changes in the catalog format.

When an end user adds structure to a document, FrameMaker does not normally identify format overrides. You can have FrameMaker flag all element instances in the document that have overrides so that the user can find the overrides and decide how to handle them in a structured context.

To flag format overrides, add the rule `flag paragraph format overrides` or `flag character format overrides` to the first column of the table. (The rule is case-insensitive.) This is a general instruction for the table, so you do not add anything to the second and third columns. For example:

Wrap this object	In this element
flag paragraph format overrides	
flag character format overrides	

At each element instance that has an override in the document, FrameMaker adds an attribute called `Override` with the value `Yes`.

Wrapping untagged formatted text

It is possible for someone to format a text range by applying commands from the Font, Size, and Style submenus in the Format menu—and not use a character format at all. This leaves the text formatted but without a tag that you can refer to in your conversion table.

You can have FrameMaker find text that has been formatted with the submenu commands and wrap it in a "catch-all" element. After adding structure to a document, the end user will probably want to look at these instances and change them to other elements (such as `Emphasis`) that more specifically describe the type of formatting.

To wrap untagged formatted text, add the rule `untagged character formatting` to the first column of the table and add an element to the second column. (The rule is case-insensitive.) For example:

Wrap this object	**In this element**
untagged character formatting	UntaggedText

This might also be useful while you are developing a conversion table. You can add structure to a sample document with this rule to see if the document has any untagged formatting.

Nesting object elements

Typically, a non-paragraph object such as a table or graphic is wrapped in an object element and then wrapped in a paragraph element. You can also wrap the object in more than one level below the paragraph. Sometimes you need to do this to conform to a DTD that requires more hierarchy, or you may just want to be able to use two objects together.

To nest object elements in a paragraph, define each mapping in a separate rule in the table. For example:

Wrap this object	**In this element**
M:Index	Index
G:	Graphic
E:Index & E:Figure	Figure

In the example above, the rules wrap an index marker in an `Index` element and a graphic anchor in a `Graphic` element, and then they wrap the two elements together in a `Figure`

text range element. This way, the graphics in a structured document will automatically have a marker identifying a location to be included in an index.

Building table structure from paragraph format tags

When FrameMaker adds structure to tables, it normally wraps all instances of a table part in the same kind of element and uses a default name for the element, such as CELL, ROW, HEADING, or BODY. You can override the default name by providing a different element tag in the second column of the conversion table.

If you want to have more than one kind of element for a particular table part, you can build the structure up from the format tags used in the cells or titles. This allows you to distinguish between different formatting used in different instances of a single table part. For example, a table may have a few special body rows with italicized text that marks divisions in the table. Or a table may have two titles, one of them a subtitle in a different font size.

To build table structure from paragraph format tags, for each cell or title rule use the TC: or TT: type identifier followed by the P: identifier and a format tag in the first column of the table. For example:

Wrap this object	In this element
TC: P:DividerCell	DividerCell
TC: P:BodyCell	BodyCell
TR:DividerCell+	ROW
TR:BodyCell+	ROW
TB:Row+	BODY

In the example above, the rules map cells that use a DividerCell paragraph format in an element called DividerCell and map cells that use a BodyCell paragraph format in an element called BodyCell. Then they wrap both kinds of cell elements in the same default ROW element and continue the wrapping normally.

Testing and correcting a conversion table

You should test and correct a conversion table as you develop it. To do this, prepare a sample document that represents the type of documents the table will apply to, and use the conversion table to add structure to the sample. Make sure your sample document has all of the document objects that the final documents may contain.

When a structure command reads a conversion table, it identifies any syntax errors in the rules and displays the errors in a log file. Correct the table and test it again until no more errors are found.

You may find it helpful to wrap only document objects for your first testing pass, without wrapping in higher levels of hierarchy. When you're sure that the rules for wrapping individual objects are correct, start writing and testing the rules to wrap elements and sequences in parent elements.

B The CALS Table Model

The CALS table model is a specific set of element and attribute declarations for defining tables, defined in “Markup Requirements and Generic Style Specification for Electronic Printed Output and Exchange of Text,” MIL-M-28001B. If your markup documents use these elements and attributes or some simple variations of them, FrameMaker can translate them to tables and table parts without the assistance of read/write rules. The CALS model can be interpreted in various ways. This appendix describes the CALS elements and attributes as they are interpreted by FrameMaker.

Some attributes are common to several elements in the description of the table. In these cases, attribute values are inherited in the element hierarchy. The values of attributes associated with <colspec> and <spanspec> elements act as though they were on the parent element for inheritance purposes. This is, if a <tgroup> element has two <colspec> child elements and a <thead> child element, the attributes of the <colspec> elements apply to the <thead> element unless that element has its own <colspec> elements with attribute values that override the inherited ones. If you want to change how FrameMaker processes any attribute of a <colspec> or <spanspec> element, you refer to the attribute as a formatting property.

In the CALS model, the <table> element has an <orient> attribute. This attribute is not supported in FrameMaker, because there is no way in a FrameMaker table to specify orientation on the page.

FrameMaker properties that DO NOT have corresponding CALS attributes

FrameMaker Property	For FrameMaker Elements of Type	Corresponding CALS Attribute
column widths	table (CALS: tgroup)	(none)

Column widths: Width of successive columns in the table. Each value is either an absolute width or a width proportional to the size of the entire table. If proportional widths are used, the CALS -pgwide- attribute determines the table width. For example, to specify that the first two columns are each one-quarter the size of the table, and the third column is half the size of the table, you could write a rule to specify your column widths as “25* 25* 50*”. Valid units and abbreviations for the “column width” formatting property are:

Unit	Abbreviation
centimeter	cm
cicero	cc
didot	dd
inch	in (in FrameMaker dialog boxes, " is also used, but not for "column width" formatting property)
millimeter	mm
pica	pc (or pi)
point	pt

FrameMaker Property	For FrameMaker Elements of Type	Corresponding CALS Attribute
maximum height	row	(none)

Maximum height of a row in a table.

minimum height	row	(none)

Minimum height of a row in a table.

row type	row	(none)

Whether the associated table row is a heading, footing, or body row, or the associated table cell occurs in a row of that type.

horizontal straddle	cell	(none)

How many columns this straddle cell spans

vertical straddle	cell	(none)

How many rows this straddled cell spans

Element and attribute definition list declarations

The element and attribute declarations as used by FrameMaker are as follows:

```
<!ENTITY % yesorno "NUMBER">
```

```
<!ELEMENT table - - (title?, tgroup+)>
<!ATTLIST table
 colsep   %yesorno; #IMPLIED
   frame    (all|top|bottom|topbot|sides|none) #IMPLIED
   pgwide   %yesorno; #IMPLIED
   rowsep   %yesorno; #IMPLIED
   tabstyle NMTOKEN   #IMPLIED
>

<!ELEMENT title - - (#PCDATA)>

<!ELEMENT tgroup - O (colspec*, spanspec*, thead?, tfoot?, tbody)>
<!ATTLIST tgroup
   align       (left|center|right|justify|char) #IMPLIED
   char        CDATA     #IMPLIED
   charoff     NUTOKEN   #IMPLIED
   colsep      %yesorno; #IMPLIED
   cols        NUMBER    #REQUIRED
   rowsep      %yesorno; #IMPLIED
   tgroupstyle NMTOKEN   #IMPLIED
>

<!ELEMENT colspec - O EMPTY>
<!ATTLIST colspec
   align    (left|center|right|justify|char) #IMPLIED
   char     CDATA     #IMPLIED
   charoff  NUTOKEN   #IMPLIED
   colname  NMTOKEN   #IMPLIED
   colnum   NUMBER    #IMPLIED
   colsep   %yesorno; #IMPLIED
   colwidth CDATA     #IMPLIED
   rowsep   %yesorno; #IMPLIED
>

<!ELEMENT spanspec - O EMPTY>
<!ATTLIST spanspec
   align    (left|center|right|justify|char) #IMPLIED
   char     CDATA     #IMPLIED
   charoff  NUTOKEN   #IMPLIED
   colsep   %yesorno; #IMPLIED
   nameend  NMTOKEN   #REQUIRED
   namest   NMTOKEN   #REQUIRED
   rowsep   %yesorno; #IMPLIED
   spanname NMTOKEN   #REQUIRED
>
```

```
<!ELEMENT thead - O (colspec*, row+)>
<!ATTLIST thead
   valign (top|middle|bottom) "bottom"
>

<!ELEMENT tfoot - O (colspec*, row+)>
<!ATTLIST tfoot
   valign (top|middle|bottom) "top"
>

<!ELEMENT tbody - O (row+)>
<!ATTLIST tbody
   valign (top|middle|bottom) "top"
>

<!ELEMENT row - O (entry+)>
<!ATTLIST row
   rowsep %yesorno;           #IMPLIED
   valign (top|middle|bottom) "top"
>

<!ELEMENT entry - O (#PCDATA)>
<!ATTLIST entry
   align    (left|center|right|justify|char) #IMPLIED
   char     CDATA     #IMPLIED
   charoff  NUTOKEN   #IMPLIED
   colname  NMTOKEN   #IMPLIED
   colsep   %yesorno; #IMPLIED
   morerows NUMBER    #IMPLIED
   nameend  NMTOKEN   #IMPLIED
   namest   NMTOKEN   #IMPLIED
   rotate   %yesorno; #IMPLIED
   rowsep   %yesorno; #IMPLIED
   spanname NMTOKEN   #IMPLIED
   valign   (top|middle|bottom) #IMPLIED
>
```

Element structure

A CALS table has an optional title followed by one or more `tgroup` elements. This allows, for example, different portions of one table to have different numbers of columns. In practice, most CALS tables have a single `tgroup` element. The `tgroup` element is the major portion of the table. It has several optional parts: multiple `colspec` and `spanspec` elements followed by (at most) one heading and one footing element. The only required sub-element of a `tgroup` element is its body. Unlike the FrameMaker model of table structure, the CALS model has its `tgroup` element appearing after the footing element.

The colspec empty element has attributes describing characteristics of a table column. The spanspec empty element has attributes describing straddling characteristics of a portion of a table. These elements have no counterpart in FrameMaker. They exist only to have their attribute values specify information about other elements in the table.

The thead and tfoot heading and footing elements contain their own optional colspec elements followed by one or more rows.

The tbody element contains one or more rows.

As supported by FrameMaker, a table row consists of a set of cells in entry elements, each of which can contain only text. Readers familiar with the CALS model may notice that these declarations do not include the entrytbl element which supports creating tables within tables. FrameMaker does not allow tables within tables, so does not support this element.

Attribute structure

Elements in the CALS table model use attributes to describe properties of the table such as cell alignment or straddling behavior. For information on the meaning of the CALS attributes, see "Formatting properties for tables" on page 276.

Inheriting attribute values

Some attributes are common to several elements in the description of a table. In these cases, attribute values are inherited in the element hierarchy. The values of attributes associated with colspec and spanspec elements act as though they were on the parent element for inheritance purposes. That is, if a tgroup element has two colspec child elements and a thead child element, the attributes of the colspec elements apply to the thead element unless that element has its own colspec elements with attribute values that override the inherited ones.

Orient attribute

In the CALS model, the table element has an orient attribute. This attribute is not supported in FrameMaker, because there is no way in a FrameMaker table to specify orientation on the page.

Straddling attributes

A spanspec element describes a column range so that a straddle cell can describe which columns it spans by referencing a spanspec through its spanname attribute.

An entry element specifies which columns it occupies by one of three methods:

- Using the namest and nameend attributes to reference columns explicitly. The namest attribute indicates the first column in the straddle; the nameend attribute indicates the last column.

- Using the `spanname` attribute as an indirect reference to the columns.
- Using the `colname` attribute (for a non-straddled cell).

C Read/Write Rules for CALS Table Model

By default, FrameMaker can read and write CALS tables without your intervention. For information on what it does by default and how you can change that behavior with read/write rules, see Chapter 18, "Translating Tables." FrameMaker does not use read/write rules to implement its default interpretation of CALS tables. However, to help your understanding of the default interpretation, this appendix contains a set of rules that encapsulate the software's default behavior for CALS tables.

As described in Chapter 18, "Translating Tables," the software's default behavior is different depending on whether the `table` element is a container element or a table element in FrameMaker. The only difference is what type of element `table` becomes and what happens to the `tgroup` element. All other elements and attributes always translate in the same way.

```
element "table" {
 /* If table is a container element, use this subrule: */
   is fm element;

 /* If table is a table element, use this subrule: */
   is fm table element;

 /* The rest of the subrules for table are always applicable. */
   attribute "tabstyle" is fm property table format;
   attribute "tocentry" is fm attribute;
   attribute "frame"
   {
      is fm property table border ruling;
      value "top"    is fm property value top;
      value "bottom" is fm property value bottom;
      value "topbot" is fm property value top and bottom;
      value "all"    is fm property value all;
      value "sides"  is fm property value sides;
      value "none"   is fm property value none;
   }
   attribute "colsep" is fm property column ruling;
   attribute "rowsep" is fm property row ruling;
   attribute "orient" is fm attribute;
   attribute "pgwide" is fm property page wide;
}
```

```
element "tgroup"
{
   /* If table is a container element, use this subrule: */
   is fm table element;

   /* If table is a table element, use this subrule: */
   unwrap;

   /*The rest of the subrules for tgroup are always applicable.*/
   attribute "cols"        is fm property columns;
   attribute "tgroupstyle" is fm property table format;
   attribute "colsep"      is fm property column ruling;
   attribute "rowsep"      is fm property row ruling;
   attribute "align"       is fm attribute;
   attribute "charoff"     is fm attribute;
   attribute "char"        is fm attribute;
}

element "colspec"
{
   is fm colspec;
   attribute "colnum"   is fm property column number;
   attribute "colname"  is fm property column name;
   attribute "align"    is fm property cell alignment type;
   attribute "charoff"  is fm property cell alignment offset;
   attribute "char"     is fm property cell alignment character;
   attribute "colwidth" is fm property column width;
   attribute "colsep"   is fm property column ruling;
   attribute "rowsep"   is fm property row ruling;
}

element "spanspec"
{
   is fm spanspec;
   attribute "spanname" is fm property span name;
   attribute "namest"   is fm property start column name;
   attribute "nameend"  is fm property end column name;
   attribute "align"    is fm property cell alignment type;
   attribute "charoff"  is fm property cell alignment offset;
   attribute "char"     is fm property cell alignment character;
   attribute "colsep"   is fm property column ruling;
   attribute "rowsep"   is fm property row ruling;
}
```

```
element "thead"
{
   is fm table heading element;
   attribute "valign" is fm attribute;
}

element "tfoot"
{
   is fm table footing element;
   attribute "valign" is fm attribute;
}

element "tbody"
{
   is fm table body element;
   attribute "valign" is fm attribute;
}

element "row"
{
   is fm table row element;
   attribute "valign" is fm attribute;
   attribute "rowsep" is fm property row ruling;
}

element "entry"
{
   is fm table cell element;
   attribute "colname"  is fm property column name;
   attribute "namest"   is fm property start column name;
   attribute "nameend"  is fm property end column name;
   attribute "spanname" is fm property span name;
   attribute "morerows" is fm property more rows;
   attribute "colsep"   is fm property column ruling;
   attribute "rowsep"   is fm property row ruling;
   attribute "rotate"   is fm property rotate;
   attribute "valign"   is fm attribute;
   attribute "align"    is fm attribute;
   attribute "charoff"  is fm attribute;
   attribute "char"     is fm attribute;
}
```

C

D SGML Declaration

To be complete, an SGML document must start with an SGML declaration. This appendix contains the text of the SGML declaration used by FrameMaker when you do not supply one. It also describes the variants of the concrete syntax that you can use in your SGML declaration and unsupported optional SGML features.

> **XML:** The XML specification states that XML must use a specific SGML declaration. This appendix pertains only to SGML structure applications. If you are only working with XML markup, you may skip this appendix.

When you import an SGML document, FrameMaker first searches for the declaration in the SGML document. If the software doesn't find the declaration there, it looks for an SGML declaration specified by your SGML application definition. If your definition does not specify an SGML declaration, then the software uses the declaration described below.

When you export a FrameMaker document to SGML, FrameMaker first tries to use an SGML declaration you specified by your application. If you haven't specified one, it uses the SGML declaration described below.

For information on how to specify an SGML declaration as part of an application, see "Application definition file" on page 50.

Text of the default SGML declaration

The SGML declaration provided by FrameMaker uses ISO Latin-1 as the character set, the reference concrete syntax, and the reference capacity set. The declaration enables the optional features `OMITTAG`, `SHORTTAG`, and `FORMAL`.

For information on the default translation between the FrameMaker and ISO Latin-1 character sets, see Appendix E, "Character Set Mapping." For information on using other ISO character sets, see Appendix F, "ISO Public Entities."

The text of the default SGML declaration is as follows:

```
<!SGML "ISO 8879:1986"

CHARSET

   BASESET "ISO Registration Number 100//CHARSET ECMA-94 Right
Part of Latin Alphabet Nr. 1//ESC 2/13 4/1"
```

```
   DESCSET
      0 9 UNUSED
      9 2 9
      11 2 UNUSED
      13 1 13
      14 18 UNUSED
      32 95 32
      127 1 UNUSED
      128 127 128
      255 1 UNUSED

CAPACITY
      PUBLIC "ISO 8879:1986//CAPACITY Reference//EN"

SCOPE DOCUMENT

SYNTAX

   SHUNCHAR         0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
19 20
            21 22 23 24 25 26 27 28 29 30 31 127 255

   BASESET "ISO Registration Number 100//CHARSET ECMA-94 Right
Part of Latin Alphabet Nr. 1//ESC 2/13 4/1"

    DESCSET 0 256 0

   FUNCTION RE 13
            RS 10
            SPACE 32
            TAB SEPCHAR 9

   NAMING         LCNMSTRT ""
            UCNMSTRT ""
            LCNMCHAR "&#45;&#46;"
            UCNMCHAR "&#45;&#46;"
            NAMECASE
               GENERAL YES
               ENTITY NO

   DELIM      GENERAL SGMLREF
            SHORTREF SGMLREF

   NAMES SGMLREF

   QUANTITY SGMLREF

FEATURES
```

```
    MINIMIZE DATATAG NO
             OMITTAG YES
             RANK NO
             SHORTTAG YES

    LINK      SIMPLE NO
          IMPLICIT NO
          EXPLICIT NO

    OTHER CONCUR NO
          SUBDOC NO
          FORMAL YES

APPINFO NONE

>
```

SGML concrete syntax variants

The SGML parser used by FrameMaker allows these modifications to the SGML reference concrete syntax:

- The NAMECASE parameter of the SGML declaration can be changed. The default settings below specify that general names are *not* case sensitive (YES), and entity names *are* case sensitive (NO):

  ```
  NAMECASE
           GENERAL  YES
           ENTITY   NO
  ```

- Reserved names can be changed.
- Short references can, but need not, be used. If they are used, the only possible short reference delimiter set is that of the reference concrete syntax.
- The value for the NAMELEN quantity can be increased up to 239.
- The values for the following quantities can be increased, but not to more than 30 times their value in the reference concrete syntax:

  ```
  ATTCNT
  ATTSPLEN
  BSEQLEN
  ENTLVL
  LITLEN
  PILEN
  TAGLEN
  TAGLVL
  ```

- The following quantities can be increased up to 253:

  GRPCNT
  GRPGTCNT
  GRPLVL

No SGML read/write rules are needed to provide for variant concrete syntaxes. FrameMaker obtains the information from the SGML declaration.

The concrete syntax declared in the SGML declaration must be used for the entire document; if a variant concrete syntax is declared, the reference concrete syntax cannot be used in the prolog. Thus, the concrete syntax scope parameter must be:

```
SCOPE DOCUMENT
```

Unsupported optional SGML features

The SGML standard defines some features as optional, meaning that a specific implementation does not have to accommodate these features to be considered a conforming SGML system.

The following optional SGML features are not supported by FrameMaker:

- DATATAG
- RANK
- LINK
- SUBDOC
- CONCUR

Your DTD and SGML documents cannot use any of these features. If they do, the FrameMaker signals an error and terminates processing. You cannot change this behavior by providing an SGML API client.

E Character Set Mapping

FrameMaker writes SGML documents using the ISO Latin-1 character set. This character set differs from FrameMaker's character set. Consequently, the software uses a default character set mapping to translate between the character sets.

> **XML:** The XML specification allows for UNICODE in content and in markup tokens, so the use of ISO character sets is not necessary. FrameMaker supports the full range of UNICODE in the content of an XML document, and offers limited support of characters in markup tokens. For more information, see "Supported characters in element and attribute names" on page 26.
>
> If you are only working with XML markup, you may skip this appendix.

FrameMaker includes copies of other ISO public entity sets and provides rules to handle them for your application. For information on how FrameMaker supports ISO public entities, see Appendix F, "ISO Public Entities."

This appendix describes the default mapping between the FrameMaker character set and the ISO Latin-1 character set. You can change this mapping by using the `character map` rule as described in "character map" on page 369.

To determine the mapping for a particular character, use the table on the next page as follows:

- For a character in the ISO Latin-1 character set, find the hexadecimal character code for the character of interest in the leftmost column. Read the mapping in the column labelled "Mapping from ISO Latin-1 to FrameMaker." The entry on the left side of the equal sign is the ISO Latin-1 character code. The entry on the right side of the equal sign is the character's translation in FrameMaker. For example, the character code `\xA1` has the entry:

  ```
  \xA1 = \xC1
  ```

  This means that the ISO Latin-1 character `\xA1` translates to the FrameMaker character `\xC1`.

- For a character in the FrameMaker character set, find the hexadecimal character code for the character of interest in the leftmost column. Read the mapping in the column labelled "Mapping from FrameMaker to ISO Latin-1." The entry on the right side of the equal sign is the FrameMaker character code. The entry on the left side of the equal sign is the character's translation in ISO Latin-1. For example, the character code `\x10` has the entry:

  ```
  \x20 = \x10
  ```

This means that the FrameMaker character \x10 translates to the ISO Latin-1 character \x20.

- If there is no row corresponding to a character code, then that character code is the same in both character sets.

Character code	Mapping from ISO Latin-1 to FrameMaker	Mapping from FrameMaker to ISO Latin-1
\x00	\x00 = trap	trap = \x00
\x01	\x01 = trap	trap = \x01
\x02	\x02 = trap	trap = \x02
\x03	\x03 = trap	trap = \x03
\x04	\x04 = trap	trap = \x04
\x05	\x05 = trap	trap = \x05
\x06	\x06 = trap	trap = \x06
\x07	\x07 = trap	trap = \x07
\x08	\x08 = trap	\x09 = \x08
\x09	\x09 = \x08	\x0A = \x09
\x0A	\x0A = \x0A	\x0A = \x0A
\x0B	\x0B = trap	trap = \x0B
\x0C	\x0C = trap	trap = \x0C
\x0D	\x0D = trap	trap = \x0D
\x0E	\x0E = trap	trap = \x0E
\x0F	\x0F = trap	trap = \x0F
\x10	\x10 = trap	\x20 = \x10
\x11	\x11 = trap	\x20 = \x11
\x12	\x12 = trap	\x20 = \x12
\x13	\x13 = trap	\x20 = \x13
\x14	\x14 = trap	\x20 = \x14
\x15	\x15 = trap	\x2D = \x15
\x16	\x16 = trap	trap = \x16
\x17	\x17 = trap	trap = \x17
\x18	\x18 = trap	trap = \x18
\x19	\x19 = trap	trap = \x19
\x1A	\x1A = trap	trap = \x1A
\x1B	\x1B = trap	trap = \x1B

Character code	Mapping from ISO Latin-1 to FrameMaker	Mapping from FrameMaker to ISO Latin-1
\x1C	\x1C = trap	trap = \x1C
\x1D	\x1D = trap	trap = \x1D
\x1E	\x1E = trap	trap = \x1E
\x1F	\x1F = trap	trap = \x1F
\x7F	\x7F = trap	trap = \x7F
\x80	\x80 = trap	\xC4 = \x80
\x81	\x81 = trap	\xC5 = \x81
\x82	\x82 = trap	\xC7 = \x82
\x83	\x83 = trap	\xC9 = \x83
\x84	\x84 = trap	\xD1 = \x84
\x85	\x85 = trap	\xD6 = \x85
\x86	\x86 = trap	\xDC = \x86
\x87	\x87 = trap	\xE1 = \x87
\x88	\x88 = trap	\xE0 = \x88
\x89	\x89 = trap	\xE2 = \x89
\x8A	\x8A = trap	\xE4 = \x8A
\x8B	\x8B = trap	\xE3 = \x8B
\x8C	\x8C = trap	\xE5 = \x8C
\x8D	\x8D = trap	\xE7 = \x8D
\x8E	\x8E = trap	\xE9 = \x8E
\x8F	\x8F = trap	\xE8 = \x8F
\x90	\x90 = trap	\xEA = \x90
\x91	\x91 = trap	\xEB = \x91
\x92	\x92 = trap	\xED = \x92
\x93	\x93 = trap	\xEC = \x93
\x94	\x94 = trap	\xEE = \x94
\x95	\x95 = trap	\xEF = \x95
\x96	\x96 = trap	\xF1 = \x96
\x97	\x97 = trap	\xF3 = \x97
\x98	\x98 = trap	\xF2 = \x98
\x99	\x99 = trap	\xF4 = \x99

Character code	Mapping from ISO Latin-1 to FrameMaker	Mapping from FrameMaker to ISO Latin-1
\x9A	\x9A = trap	\xF6 = \x9A
\x9B	\x9B = trap	\xF5 = \x9B
\x9C	\x9C = trap	\xFA = \x9C
\x9D	\x9D = trap	\xF9 = \x9D
\x9E	\x9E = trap	\xFB = \x9E
\x9F	\x9F = trap	\xFC = \x9F
\xA0	\xA0 = trap	trap = \xA0
\xA1	\xA1 = \xC1	trap = \xA1
\xA2	\xA2 = \xA2	\xA2 = \xA2
\xA3	\xA3 = \xA3	\xA3 = \xA3
\xA4	\xA4 = \xDB	\xA7 = \xA4
\xA5	\xA5 = \xB4	\xB7 = \xA5
\xA6	\xA6 = \x7C	\xB6 = \xA6
\xA7	\xA7 = \xA4	\xDF = \xA7
\xA8	\xA8 = \xAC	\xAE = \xA8
\xA9	\xA9 = \xA9	\xA9 = \xA9
\xAA	\xAA = \xBB	trap = \xAA
\xAB	\xAB = \xC7	\xB4 = \xAB
\xAC	\xAC = \xC2	\xA8 = \xAC
\xAD	\xAD = \x2D	trap = \xAD
\xAE	\xAE = \xA8	\xC6 = \xAE
\xAF	\xAF = \xF8	\xD8 = \xAF
\xB0	\xB0 = \xFB	trap = \xB0
\xB1	\xB1 = trap	trap = \xB1
\xB2	\xB2 = trap	trap = \xB2
\xB3	\xB3 = trap	trap = \xB3
\xB4	\xB4 = \xAB	\xA5 = \xB4
\xB5	\xB5 = trap	trap = \xB5
\xB6	\xB6 = \xA6	trap = \xB6
\xB7	\xB7 = \xA5	trap = \xB7
\xB8	\xB8 = \xFC	trap = \xB8
\xB9	\xB9 = trap	trap = \xB9

Character code	Mapping from ISO Latin-1 to FrameMaker	Mapping from FrameMaker to ISO Latin-1
\xBA	\xBA = \xBC	trap = \xBA
\xBB	\xBB = \xC8	\xAA = \xBB
\xBC	\xBC = trap	\xBA = \xBC
\xBD	\xBD = trap	trap = \xBD
\xBE	\xBE = trap	\xE6 = \xBE
\xBF	\xBF = \xC0	\xF8 = \xBF
\xC0	\xC0 = \xCB	\xBF = \xC0
\xC1	\xC1 = \xE7	\xA1 = \xC1
\xC2	\xC2 = \xE5	\xAC = \xC2
\xC3	\xC3 = \xCC	trap = \xC3
\xC4	\xC4 = \x80	trap = \xC4
\xC5	\xC5 = \x81	trap = \xC5
\xC6	\xC6 = \xAE	trap = \xC6
\xC7	\xC7 = \x82	\xAB = \xC7
\xC8	\xC8 = \xE9	\xBB = \xC8
\xC9	\xC9 = \x83	trap = \xC9
\xCA	\xCA = \xE6	trap = \xCA
\xCB	\xCB = \xE8	\xC0 = \xCB
\xCC	\xCC = \xED	\xC3 = \xCC
\xCD	\xCD = \xEA	\xD5 = \xCD
\xCE	\xCE = \xEB	trap = \xCE
\xCF	\xCF = \xEC	trap = \xCF
\xD0	\xD0 = trap	\x2D = \xD0
\xD1	\xD1 = \x84	\x2D = \xD1
\xD2	\xD2 = \xF1	\x22 = \xD2
\xD3	\xD3 = \xEE	\x22 = \xD3
\xD4	\xD4 = \xEF	\x60 = \xD4
\xD5	\xD5 = \xCD	\x27 = \xD5
\xD6	\xD6 = \x85	trap = \xD6
\xD7	\xD7 = trap	trap = \xD7
\xD8	\xD8 = \xAF	\xFF = \xD8
\xD9	\xD9 = \xF4	trap = \xD9

Character code	Mapping from ISO Latin-1 to FrameMaker	Mapping from FrameMaker to ISO Latin-1
\xDA	\xDA = \xF2	\x2F = \xDA
\xDB	\xDB = \xF3	\xA4 = \xDB
\xDC	\xDC = \x86	trap = \xDC
\xDD	\xDD = trap	trap = \xDD
\xDE	\xDE = trap	trap = \xDE
\xDF	\xDF = \xA7	trap = \xDF
\xE0	\xE0 = \x88	trap = \xE0
\xE1	\xE1 = \x87	\xB7 = \xE1
\xE2	\xE2 = \x89	\x2C = \xE2
\xE3	\xE3 = \x8B	trap = \xE3
\xE4	\xE4 = \x8A	trap = \xE4
\xE5	\xE5 = \x8C	\xC2 = \xE5
\xE6	\xE6 = \xBE	\xCA = \xE6
\xE7	\xE7 = \x8D	\xC1 = \xE7
\xE8	\xE8 = \x8F	\xCB = \xE8
\xE9	\xE9 = \x8E	\xC8 = \xE9
\xEA	\xEA = \x90	\xCD = \xEA
\xEB	\xEB = \x91	\xCE = \xEB
\xEC	\xEC = \x93	\xCF = \xEC
\xED	\xED = \x92	\xCC = \xED
\xEE	\xEE = \x94	\xD3 = \xEE
\xEF	\xEF = \x95	\xD4 = \xEF
\xF0	\xF0 = trap	trap = \xF0
\xF1	\xF1 = \x96	\xD2 = \xF1
\xF2	\xF2 = \x98	\xDA = \xF2
\xF3	\xF3 = \x97	\xDB = \xF3
\xF4	\xF4 = \x99	\xD9 = \xF4
\xF5	\xF5 = \x9B	trap = \xF5
\xF6	\xF6 = \x9A	\x5E = \xF6
\xF7	\xF7 = trap	\x7E = \xF7
\xF8	\xF8 = \xBF	\xAF = \xF8
\xF9	\xF9 = \x9D	trap = \xF9

Character code	Mapping from ISO Latin-1 to FrameMaker	Mapping from FrameMaker to ISO Latin-1
\xFA	\xFA = \x9C	trap = \xFA
\xFB	\xFB = \x9E	\xB0 = \xFB
\xFC	\xFC = \x9F	\xB8 = \xFC
\xFD	\xFD = trap	trap = \xFD
\xFE	\xFE = trap	trap = \xFE
\xFF	\xFF = \xD8	trap = \xFF

F ISO Public Entities

Annex D of the SGML standard defines several sets of internal SDATA entities. Each entity represents a character; each entity set is a logical grouping of these entities. DTDs frequently include these entity sets by using parameter entity references to external entities accessed with a public identifier. People in the SGML community frequently interchange DTDs and SGML documents with such entity references and assume that the recipient can interpret the public identifiers. FrameMaker includes copies of these entity sets and makes them available using the default handling of public identifiers.

> **XML:** The XML specification does not allow SDATA entities, but it does allow UNICODE and predefined character entities for special characters. This appendix pertains only to SGML structure applications. If you are only working with XML markup, you may skip this appendix.

Because these entity sets are defined in an ISO standard and are accessed with public identifiers, they are commonly known as *ISO public entity sets*. The public entity sets fall into the following categories:

Entity set	Description
Latin alphabetic characters	Latin alphabetic characters used in Western European languages
Greek alphabetic characters	Letters of the Greek alphabet
Greek symbols	Greek character names for use as variable names in technical applications
Cyrillic alphabetic characters	Cyrillic characters used in the Russian language
Numeric and special graphic characters	Minimum data characters and reference concrete syntax characters
Diacritical mark characters	Diacritical marks
Publishing characters	Well-known publishing characters
Technical symbols	Technical symbols
Added math symbols	Mathematical symbols

If your application uses FrameMaker's support of ISO entity sets, you may want to create palettes your end user can use to enter these entities in a FrameMaker document. For information on creating these palettes, see "Facilitating entry of special characters that translate as entities" on page 268.

What you need to use ISO public entities

For your end users to use characters from the ISO public entity sets, your application needs two pieces of information for each character entity: the entity's declaration, and an SGML read/write rule that tells FrameMaker how to translate a reference to that entity in an SGML document to a character or variable in a FrameMaker document. FrameMaker provides this information in two files for each entity set.

All files used for ISO public entity sets are in the directory `$STRUCTDIR/isoents`. For information on the location of this directory on your system, see "Location of structure files" on page 49. The files for each entity set are as follows:

Entity set	Entity declaration files	Read/write rules files
Latin alphabetic characters	`isolat1.ent` `isolat2.ent`	`isolat1.rw` `isolat2.rw`
Greek alphabetic characters	`isogrk1.ent` `isogrk2.ent`	`isogrk1.rw` `isogrk2.rw`
Greek symbols	`isogrk3.ent` `isogrk4.ent`	`isogrk3.rw` `isogrk4.rw`
Cyrillic alphabetic characters	`isocyr1.ent` `isocyr2.ent`	`isocyr1.rw` `isocyr2.rw`
Numeric and special graphic characters	`isonum.ent`	`isonum.rw`
Diacritical mark characters	`isodia.ent`	`isodia.rw`
Publishing characters	`isopub.ent`	`isopub.rw`
Technical symbols	`isobox.ent` `isotech.ent`	`isobox.rw` `isotech.rw`

Added math symbols	`isoamso.ent`
	`isoamsb.ent`
	`isoamsr.ent`
	`isoamsn.ent`
	`isoamsa.ent`
	`isoamsc.ent`
		`isoamso.rw`
		`isoamsb.rw`
		`isoamsr.rw`
		`isoamsn.rw`
		`isoamsa.rw`
		`isoamsc.rw`

Entity declaration files

Each entity declaration file starts with two comment declarations that suggest both the public identifier and the entity name by which to identify the entity set. For the ISO Latin-1 entity set, these comments are:

```
<!-- (C) International Organization for Standardization 1986
      Permission to copy in any form is granted for use with
      conforming SGML systems and applications as defined in
      ISO 8879, provided this notice is included in all copies.
-->
<!-- Character entity set. Typical invocation:
      <!ENTITY % ISOlat1 PUBLIC
         "ISO 8879-1986//ENTITIES Added Latin 1//EN">
      %ISOlat1;
-->
```

After the initial comments, an entity declaration file consists of a sequence of entity declarations. For example, the first few entity declarations for ISO Latin-1 are as follows:

```
<!ENTITY aacute SDATA "[aacute]"--=small a, acute accent-->
<!ENTITY Aacute SDATA "[Aacute]"--=capital A, acute accent-->
<!ENTITY acirc SDATA "[acirc ]"--=small a, circumflex accent-->
<!ENTITY Acirc SDATA "[Acirc ]"--=capital A, circumflex accent-->
<!ENTITY agrave SDATA "[agrave]"--=small a, grave accent-->
<!ENTITY Agrave SDATA "[Agrave]"--=capital A, grave accent-->
<!ENTITY aring SDATA "[aring ]"--=small a, ring-->
<!ENTITY Aring SDATA "[Aring ]"--=capital A, ring-->
```

You should never modify these files, because they provide the standard ISO public entity declarations.

If your SGML documents use the standard invocations for ISO public entity sets, you do not have to provide any information in your application definition on where to find these entities; FrameMaker finds them in the default directory. If necessary, you can provide explicit `public` statements to substitute alternative versions of the entity sets. For information on working with application definitions, see “Application definition file” on page 50.

Entity read/write rules files

FrameMaker provides read/write rules for many of the entities in the ISO public entity sets. The rules are organized in files for each public entity set. These files are not complete rules documents. Instead, they are simply lists of rules or comments explaining which entities do not have default correspondences.

You can include individual files in your application's read/write rules document by using the `#include` statement. To include the rules for all of the ISO public entity sets, use this single statement:

```
#include isoall.rw
```

To include only the ISO Latin-1 entity set, use these statements:

```
#include isolat1.rw
#include isolat2.rw
```

For more information on read/write rules files, see Chapter 14, "Read/Write Rules and Their Syntax."

Format of entity rules

By default, FrameMaker has rules for each entity that can be represented in FrameMaker using the standard FrameMaker character set, the Symbol font, or the Zapf Dingbat font and for a few (such as the fractions in `isonum`) that can be represented with a FrameMaker user variable. Entities that cannot be represented in this way do not have a default translation. Users of your application may have more fonts available. If so, you can modify these rules files to include translations for other entities.

The default rules for entities available in the default character sets or through variables differ depending on how FrameMaker translates the entity.

- If the character appears in FrameMaker's standard character set and requires no special formatting, the rule has the following form:

  ```
  entity "ename" is fm char code;
  ```

  where *ename* is the entity name and *code* is the character code. For example, the following rule is for the small letter "a" with an acute accent:

  ```
  entity "aacute" is fm char 0x87;
  ```

- If the character appears in FrameMaker's Symbol or Zapf Dingbat character set or appears in FrameMaker's standard character set, but requires special formatting, the rule has the following form:

  ```
  entity "ename" is fm char code in "fmtag";
  ```

  where *ename* is the entity name, *code* is the character code, and *fmtag* is one of the character tags defined below. For example, the following rule is for the plus-or-minus sign:

  ```
  entity "plusnm" is fm char 0xb1 in "FmSymbol";
  ```

- If the character can be represented by an FrameMaker variable, the rule has the following form:

  ```
  entity "ename" is fm variable "var";
  ```

where *ename* is the entity name and *var* is one of the FrameMaker variables defined below. For example, the following rule is for the fraction one-half:

```
entity "frac12" is fm variable "FmFrac12";
```

For details on how each entity translates into a FrameMaker document, see the individual rules files.

Character formats

As mentioned above, the rules for some character entities refer to FrameMaker character formats or variable names. FrameMaker has default definitions for these character formats:

Character format	Defined as
FmDenominator	Default font, subscripted; other characteristics As Is
FmDingbats	Zapf Dingbat font; other characteristics As Is
FmNumerator	Default font, superscripted; other characteristics As Is
FmSdata	Default font, underlined and in green; other characteristics As Is
FmSuperscript	Default font superscripted; other characteristics As Is
FmSymbol	Symbol font; other characteristics As Is
FmUnderlineSymbol	Symbol font, underlined; other characteristics As Is

Variables

FrameMaker also has default definitions for these variables:

Variable	Defined as
FmCare-of	$^{c}/_{o}$
FmEmsp13	an em space
FmFrac12	$^{1}/_{2}$
FmFrac13	$^{1}/_{3}$
FmFrac14	$^{1}/_{4}$
FmFrac15	$^{1}/_{5}$
FmFrac16	$^{1}/_{6}$
FmFrac18	$^{1}/_{8}$
FmFrac23	$^{2}/_{3}$
FmFrac25	$^{2}/_{5}$
FmFrac34	$^{3}/_{4}$
FmFrac35	$^{3}/_{5}$
FmFrac38	$^{3}/_{8}$
FmFrac45	$^{4}/_{5}$
FmFrac56	$^{5}/_{6}$

Variable	Defined as
FmFrac58	$^5/_8$
FmFrac78	$^7/_8$

Your end user's documents may not have these character formats or variables defined. When FrameMaker imports an SGML document with an entity reference that needs one of these formats or variables, it checks whether the template defined in the SGML application provides the definition. If so, it uses the information from the template. If not, it uses its own definitions, copying the definition to the appropriate catalog of the document being processed and using it to process the entity.

What happens with the declarations and rules

Your application may use some or all of the entity declarations and read/write rules provided with FrameMaker. In addition, you may choose to have different declarations or rules for some or all of the entities.

If you want to use the translations provided by FrameMaker with no changes, you do so in one of two ways.

- If your application has no other read/write rules, then you do not have to explicitly mention the rules for these entity sets. That is, if the definition of your application does not include a read/write rules file, FrameMaker behaves as though it had a rules file that included only the ISO public entity rules.
- On the other hand, if your application does have a read/write rules file, then that file must explicitly include the rules for the ISO public entity sets in which you're interested. If you want all of them, add the following line to your file:

  ```
  #include isoall.rw
  ```

  When you create a new read/write rules file, this line is automatically included.

If you want to use only the rules that FrameMaker provides, be sure that your rules file has no additional `entity` rules referring to these entities. However, you may want to have FrameMaker translate most but not all of these entities in the way it provides, while you change the behavior for some of them with rules or entity declarations. To do this, include an extra entity declaration or rule for the appropriate entities.

For example, assume the DTD for your application is called `myapp.dtd` and includes the following lines:

```
<!ENTITY % ISOlat1 PUBLIC
         "ISO 8879-1986//ENTITIES Added Latin 1//EN">
%ISOlat1;
```

Further, assume the application has no rules or has a rules document that contains the following lines:

```
#include "isolat1.rw"
#include "isolat2.rw"
```

The default version of `isolat1.rw` contains the rule:

```
entity "aacute" is fm char 0x87;
```

This translates references to the `aacute` entity as the small letter a with an acute accent. Suppose, however, that your application needs this entity, instead, to translate as a particular bitmap that you store as a reference element in the FrameMaker document template. You accomplish this by adding a rule for that entity before the #include statements, as follows:

```
entity "aacute" is fm reference element "acute-a";
```

Remember that FrameMaker uses the first rule in a rules file that applies to a particular situation. Therefore, if you use this rule, then the line in the example that includes `isolat1.rw` must occur after this rule. That is, your rules file must look like:

```
entity "aacute" is fm reference element "acute-a";
. . .
#include isolat1.rw
. . .
```

If, instead, it looks like:

```
#include isolat1.rw
. . .
entity "aacute" is fm reference element "acute-a";
. . .
```

FrameMaker finds the rule in `isolat1.rw` before your rule and use that to process references to the `aacute` entity.

FrameMaker has rules for entities in the ISO public entity sets for which there is a direct correspondence in one of its standard character sets or which can be created using a character from those character sets. It does not provide rules for entities that would require a different character set or a graphic.

If you reference an ISO public entity for which there is not a rule, the software treats it as it does any other entity that doesn't have a corresponding rule. You can change this behavior with the `entity` rule. For more information on FrameMaker's translation of entities in the absence of rules and for information on how you can modify this, see Chapter 17, "Translating Entities and Processing Instructions."

G SGML Batch Utilities for UNIX

The UNIX version of FrameMaker provides two utility programs, `fmimportsgml` and `fmexportsgml`, in addition to the utility programs defined for FrameMaker and described in the FrameMaker user's manual. You use `fmimportsgml` for batch importing of markup files and `fmexportsgml` for batch exporting of FrameMaker files to markup.

Some points to keep in mind about these programs:

- The utility programs start FrameMaker. For this reason, they can run only on a system capable of running a version FrameMaker that uses the Structured FrameMaker program interface.
- The programs overwrite existing files. If your user preferences specify creation of backup files, the programs create these files.
- The programs recognize wildcards in filenames. It is an error if a wildcard refers to multiple files where a single file is expected.
- Labeled parameters are those of the form:

  `-parameter_name` *`parameter_value`*

  All labeled parameters are optional. They can be entered in any order preceding the names of the documents to be converted.

Importing SGML documents in batch mode

The `fmimportsgml` program imports markup documents as FrameMaker documents. You call it as follows:

```
fmimportsgml
   [-dir dir]
   [-suffix suffix]
   [-flow flow]
   [-structapps fname]
   [-app app_name | -noapp]
   [-log log_file]
   [-language lang]
   [-v]
   [-h]
markup_documents
```

where:

`-dir` *dir*	names the directory where `fmimportsgml` stores output files. If this parameter is not specified, `fmimportsgml` uses the current directory.
`-suffix` *suffix*	specifies the suffix for FrameMaker documents created by `fmimportsgml`. This option does not apply when you are importing a book; in this case, the suffix is always `.book`. If this parameter is not specified, the suffix is `.fm` for a single document.
`-flow` *flow*	specifies a flow tag for the output FrameMaker document. If this parameter is present, the application definition must specify a FrameMaker template. `fmimportsgml` writes its output to the first flow with the indicated tag that occurs on a body page of the template. FrameMaker appends the translation of the markup document to the end of the content (if any) of the specified flow. It is an error if the flow doesn't exist.
`-structapps` *fname*	specifies a structure application file to use in place of `$STRUCTDIR/structapps.fm`. If this parameter is not specified, `fmimportsgml` uses the standard search path to find `structapps.fm`—for details, see "Location of structure files" on page 49.
`-app` *app_name* `-noapp`	specifies the application definition to use or specifies using no application at all. If this parameter is not specified, `fmimportsgml` uses no application.
`-log` *log_file*	specifies the name of a FrameMaker document to use as the log file. If this parameter is not present and if there are errors during conversion, `fmimportsgml` creates the log file `sgmllog.doc` in the current directory.
`-v`	displays the software version. When `-v` is specified, all other parameters are ignored.
`-h`	displays a brief description of `fmimportsgml` and its parameters. When `-h` is specified, the software ignores all other parameters. Omitting all parameters is equivalent to specifying `-h`.
markup_documents	names one or more markup documents to be imported.

For example, assume you the `/sgmlbook` directory contains the files `a.sgml`, `b.sgml`, and `c.txt` and you use this command:

```
fmimportsgml -app MyApp -dir /fmbook -suffix doc /mydocs/*.sgml
```

The software uses the MyApp application defined in `structapps.fm` to create the files `a.doc` and `b.doc` in the `/fmbook` directory.

Exporting documents as SGML in batch mode

The `fmexportsgml` program exports FrameMaker documents as markup documents. You call it as follows:

```
fmexportsgml
   [-dir dir]
   [-suffix suffix]
   [-flow flow]
   [-structapps fname]
   [-app app_name | -noapp]
   [-log log_file]
   [-v]
   [-h]
fm_documents
```

where:

`-dir` *dir*	names the directory where `fmexportsgml` stores output files. If this parameter is not specified, `fmexportsgml` uses the current directory.
`-suffix` *suffix*	specifies the suffix for markup documents created by `fmexportsgml`. If this parameter is not specified, the suffix is `.sgm`.
`-flow` *flow*	specifies a flow tag for the FrameMaker input document. `fmexportsgml` exports the first flow with the indicated tag that occurs in the FrameMaker input document; it is an error if there is no matching flow. If this parameter is not specified, FrameMaker exports the main flow of the document.
`-structapps` *fname*	specifies a structure application file to use in place of `$STRUCTDIR/structapps.fm`. If this parameter is not specified, `fmexportsgml` uses the standard search path to find `structapps.fm`—for details, see "Location of structure files" on page 49.
`-app` *app_name* `-noapp`	specifies the application definition to use or specifies using no application at all. If this parameter is not specified, `fmexportsgml` uses no application.
`-log` *log_file*	specifies the name of a FrameMaker document to use as the log file. If this parameter is not present and if there are errors during conversion, `fmexportsgml` creates the log file `sgmllog.doc` in the current directory.

`-v`	displays the software version. When `-v` is specified, all other parameters are ignored.
`-h`	displays a brief description of `fmexportsgml` and its parameters. When `-h` is specified, the software ignores all other parameters. Omitting all parameters is equivalent to specifying `-h`.
`fm_documents`	names one or more FrameMaker documents to be exported.

For example, assume you the `/fmbook` directory contains the files `a.fm`, `b.fm`, and `c.text` and you use this command:

```
fmexportsgml -noapp dir /sgmlbook -suffix sgml /mydocs/*.fm
```

The software does not use an SGML application. It creates the files `a.sgml` and `b.sgml` in the `/sgmlbook` directory.

H Developing XML or SGML Publishing Applications

Implementing an XML or SGML application in the FrameMaker publishing environment

Before you begin creating structure applications, you should understand the relationship between *markup* and FrameMaker, and the task of developing structure applications. To introduce you to the task of integrating FrameMaker in an XML or SGML publishing environment, this chapter provides:

- An explanation of the FrameMaker application philosophy
- An introduction to the technical steps in application development
- A description of some typical application development scenarios

Overview of FrameMaker Application Development

This section examines the relationship between SGML and various types of SGML applications. It also presents an overview of developing applications for FrameMaker.

Markup Applications

A markup application is an application that manages and stores critical business information in a markup format. For this document, markup refers to either XML (Extensible Markup Language) or SGML (Standardized General Markup Language). There are two important aspects of markup that make it suitable for streamlining the creation, maintenance, and delivery of critical business information:

- Open standards
- Formal structure

If you can take advantage of these aspects, you will see improvements in your publishing process, and increased accuracy in the information you deliver. Because of open standards, you can share your information with more people using a wider variety of software applications. If you take advantage of the formal structure in markup, you can develop increasingly sophisticated ways to automate your processes.

Part of the formality of markup is that it separates content from appearance. The way markup separates a document's content from its appearance and intended processing makes it a natural format for single-source documents. However, the ability to use text in different ways requires that each use be defined. In markup terminology, such a definition is called an *application*. More precisely, the SGML standard (ISO 8879, Standard

Generalized Markup Language) defines a text processing application to be “a related set of processes performed on documents of related types.” In particular, the standard states that an SGML application consists of “rules that apply SGML to a text processing application including a formal specification of the markup constructs used in the application. It can also include a definition of processing.”

This definition of an SGML application equally applies to define XML applications. According to the XML specification, XML is a subset of SGML. In that sense, nearly anything that can be said of SGML can also be said of XML. For this chapter, the term *markup* refers to XML and SGML equally, and a *markup application* is an application for either XML or SGML. When necessary, this chapter specifically refers to XML or SGML to point out differences between the two.

This means that whether you are interested in XML or SGML, a publishing system that uses this technology is a general SGML application. To avoid confusion, this document refers to a general SGML application as a *markup application*. A markup application is the sum total of the tools in your XML or SGML publishing system. These tools work together to author, process, and deliver your information in a variety of formats that can include print, PDF, HTML, or business transactions.

Understanding Markup Applications The need for markup applications is easily understood by comparing general-purpose markup tools to traditional database packages. Just as database software is used to maintain and access collections of numeric data and fixed-length text strings, a markup system is used to maintain and access document databases. Both types of systems typify powerful, flexible products that must be configured before their power can be used effectively. In particular, both tools require the user to define the data to be processed as well as how that data will be entered and accessed.

Database	Markup
Depends on a schema to define record layout	Uses a document type definition (DTD) to define structural elements and attributes
Input forms facilitate data entry	Configured XML or SGML text editor facilitates data entry
Reports present customized summaries and other views of information	Composed documents present information to end user

The *DTD* and accompanying input and processing conventions comprise the rules that define a markup application.

These ideas are easily illustrated with an example: While markup applications need not include a visual rendering of documents—consider a voice synthesizer, word count, or linguistic analysis tool—many of them do involve publishing. Publishing is the process of distributing information to the consumer. Traditional publishing involves preparation of written materials such as books, magazines, and brochures. Computerized information

processing facilitates publishing in additional media, such as CD-ROM and the Internet, as well as other applications, such as database distribution.

Often, the same information is published in multiple forms. In such cases, its appearance may change according to the purpose and capabilities of each medium. The appropriate rendering enhances the reader's comprehension of the text. Since markup prepares information for multiple presentations without making assumptions about its rendering, additional information is needed to control each rendering. This formatting information, pertinent to an application, is nevertheless outside the scope of markup itself.

Consider, for instance, a repair manual stored in markup. It may include procedures such as the following:

```
<taskmodule skill="adv">
<heading>Tuning a 5450A Widget</heading>
<warning type="1">Do not hit a widget with a hammer.
Doing so could cause explosive decompression.</warning>
<background>The 5450A Widget needs tuning on a
<emph>monthly basis</emph> to maintain optimum
performance. The procedure should take less than 15
minutes.</background>
<procedure>
<step time="1">Remove the tuning knob cover.</step>
<step time="5">Use the calibration tool to adjust the
setting to initial specifications.</step>
</procedure>
</taskmodule>
```

The markup is passed through a rendering application to present the mechanics with a formatted version:

Tuning a 5450A Widget

This procedure requires advanced certification.

Warning: Do not hit a widget with a hammer. Doing so could cause explosive decompression.

The 5450A Widget needs tuning on a *monthly basis* to maintain optimum performance. The procedure should take less than 15 minutes.

1. Remove the tuning knob cover.

2. Use the calibration tool to adjust the setting to initial specifications.

The application that prepares the formatted version applies rules indicating how the taskmodule, steps, and so on of the markup example are to appear. In this example these rules:

- Put the heading in a large, bold font.
- Generate the sentence, “This procedure requires advanced certification” from the value of the skill attribute.
- Number the steps in the `procedure`.

Notice that the values of the `time` attributes do not appear in the formatted text. The rules that drive a different application—a scheduling program, for instance—might use the `time` attribute to allocate the mechanic’s time, for example. Or it might assemble the entire maintenance procedure (possibly thousands of such maintenance cards), and add up the times for each step to estimate the maintenance schedule.

Markup Editing and Publishing Applications Two important classes of markup applications involve creating (editing) and publishing documents. Tools from different vendors take different approaches to the two applications. Editing tools are used to create markup documents; publishing tools produce formatted results from existing markup documents. For example, Datalogics’ Composer is a publishing tool that operates on completed SGML documents using *FOSI*s (Formatting Output Specification Instances) to specify page design and the visual characteristics applied to various structural elements. James Clark’s *JADE* is a non-commercial publishing tool that uses *DSSL* (Document Style and Semantic Specification Language) to specify the formatting rules of an application. Many XML rendering applications use CSS (Cascading Style Sheets) to express the visual appearance of an XML document.

Since users can best work with an editor if they have some visual indication of the document’s structure, applications for markup text editors, such as Adobe FrameMaker, Softquad’s XMetaL, and Arbortext’s Epic, also require some formatting specifications. Some editors also provide for rules to determine:

- How closely documents under development must conform to a DTD
- Options for listing available elements for the user
- Whether to automatically insert elements when the user creates a new document or inserts an element with required sub-elements
- Whether to prompt for attribute values as soon as the user creates a new element

Some tools address either the editing or the publishing application. FrameMaker is a single tool that addresses both. Although FrameMaker can be used for either one in isolation, its strengths include the ability to create new structured documents (or edit existing ones) that are ready for publishing without additional processing. This pairing allows a single set of format rules to support both tasks. Thus, less effort is required to configure FrameMaker than to configure separate editing and publishing tools.

Adobe FrameMaker Application Development

Developing a FrameMaker application shares many steps with developing a markup application for any other editing or publishing tool. This section summarizes the process, covering the steps common to other editing and publishing tools, and those unique to FrameMaker. Itemization of the various steps is preceded by a brief description of the approach to document processing that FrameMaker uses.

FrameMaker Editing Philosophy As the preceding formatted `taskmodule` example illustrates, visual clues—font variation, indentation, and numbering—help readers understand and use written information. FrameMaker is a *WYSIWYG* editor. While most *WYSIWYG* editors require authors to indicate how each part of the document is formatted, FrameMaker uses the rules in the markup application to format document components automatically in a consistent manner.

This combination of *WYSIWYG* techniques with the structured principles of markup is unique to FrameMaker. The goal in FrameMaker, is to streamline the process of creating and publishing documents through the use of markup. While authors manipulate elements and their attributes to create native markup documents, they work in a *WYSIWYG* fashion instead of directly with markup syntax.

Since the markup document model underlies edited material, the markup form of the document can be produced at any time during the *WYSIWYG* editing process. This process relies on the underlying element structure and hence has no need for the inaccurate filtering schemes that plague tools for generating markup from word processing formats.

Tasks in FrameMaker application development The major tasks involved in the development of a FrameMaker application are:

- Defining the elements that can be used in a document and the contexts in which each is permitted. This task is analogous to developing a DTD. In fact, the element definitions can be automatically derived from a DTD if one exists.
- Determining which elements correspond to special objects, such as tables, graphics, and cross-references.
- Developing format rules that define the appearance of a structural element in a particular context.
- Providing page-layout information such as running footers and headers, margins, and so on—the foundation of every *WYSIWYG* document.
- Establishing a correspondence between the markup and FrameMaker representations of a document. For most elements, this correspondence is straightforward and automatic, but FrameMaker allows for some variations. For example, terse markup names can be mapped into longer, more descriptive FrameMaker names, and variant representations of tables, graphics, and other entities can be chosen.

Each of these tasks requires planning and design before implementation, as well as testing afterward. The rest of this paper describes these steps, and the skills needed to perform them, in more detail.

The analysis phase The specific features of a FrameMaker application largely depend on the tasks it will perform:

- Will users create new content, or will the application simply format existing markup documents?
- Whether or not they require further editing, are existing markup documents to be brought into the system? What about documents that are not XML or SGML (unstructured FrameMaker documents or, perhaps, the output of a word processor)? Will such legacy documents be imported on an ongoing basis or only at the beginning of the project?
- Will finished documents be saved as markup? Will authors themselves output markup, or will they pass completed projects to a production group that performs this post-processing step?
- Does the application involve a database or document management tool? Will portions of the documents be generated automatically? Are there calculations to be performed?
- Will there be a single DTD or a family of related DTDs used for different tasks (for example, a reference DTD used for interchange with a variant authoring DTD)?

Unless a project is based on one or more existing DTDs, one of the first outputs of the analysis phase is definition of the document structures to be used. The definition process usually involves inspecting numerous examples of typical documents. Even when there is a DTD to start from, the analysis phase is necessary to produce at least tentative answers to questions such as the previously listed ones.

Defining Structure A primary goal of the analysis phase is defining the structures that the end user will manipulate within FrameMaker. The bulk of this effort is often the creation of a DTD. When the project begins with an existing DTD, the DTD provides a foundation for the structure definition. FrameMaker, in fact, can use the DTD directly. Nevertheless, a pre-existing DTD does not eliminate the need for analysis and definition. As Eve Maler and Jeanne El Andaloussi explain in *Developing SGML DTDs: From Text to Model to Markup*, many projects use several related DTDs. If the existing DTD is intended for interchange, the editing environment can often be made more productive by creating an editing DTD. Several examples from the widely known DocBook DTD used for computer software and hardware documentation illustrate the types of changes that might be made:

- Changing some generic identifiers, attribute names, or attribute values to reflect established terminology within the organization.

  For example, DocBook uses the generic identifier `CiteTitle` for cited titles. If authors are accustomed to tagging such titles as `Book`, an application might continue to use the name `Book` during editing, but automatically convert `Book` to and from `CiteTitle` when reading or writing the markup form of the document.

- Omitting unnecessary elements and attributes.

  DocBook defines about 300 elements. Many organizations use only a small fraction of these. There is no need to make the remainder available in an interactive application. FrameMaker displays a catalog of the elements that are valid at a given point in a

document. Removing unnecessary elements from the authoring DTD means the catalog displays only the valid elements that an organization might actually use in a particular context and avoids overwhelming the user with a much larger set of valid DocBook elements.

- Providing alternate structures.

  For example, DocBook provides separate element types—`Sect1`, `Sect2`, and so on—for different levels of sections and subsections. Defining a single `Section` element to be used at all levels (with software determining the context and applying an appropriate heading style) simplifies the task of rearranging material and hence provides a better environment for authors who may need to reorganize a document. Again, `Sections` can be automatically translated to and from the appropriate `Sect1`, `Sect2`, or other DocBook section element.

- Simplifying complex structures that are not needed by an organization.

  DocBook, for instance, provides a very rich structure for reporting error messages. If only a few of the many possibilities will actually be used, you can edit the DTD to eliminate unnecessary ones.

In addition to providing and editing the DTD, the analysis and definition of structure includes planning for the use of tables and graphics. Many DTDs provide elements that are clearly intended for such inherently visual objects. In other cases, however, such a representation of an element results from the formatting rules of the application. Consider again, for instance, the repair procedure on page 523. The formatted version did not display timing information for the procedure steps. This alternate version formats the procedure in a three-column table, showing the duration of each step:

Tuning a 5450A Widget

This procedure requires advanced certification.

Do not hit a widget with a hammer. Doing so could cause explosive decompression.

Step	Duration	Description
1	1 minute	Remove the tuning knob cover.
2	5 minutes	Use the calibration tool to adjust the setting to initial specifications.

Thus, defining structure involves defining the family of DTDs to be used in the project as well as deciding in general terms how the various elements will be used.

The Design Phase The analysis phase of the project evolves into a design phase with two major goals: defining the desired results as well as the best way to accomplish those results. Defining the desired results includes the graphic design of the completed documents: page layouts as well as visual characteristics to be assigned to each structural element. Even when there is a rich body of sample documents and a tradition of consistent

use throughout an organization, reexamination and systematic inspection may reveal unexpected inconsistencies and suggest new approaches. As a result, analysis frequently results in changes to existing processes.

Defining the best way to accomplish the desired results involves planning how to use FrameMaker—and any other relevant software—to meet the project goals. While this step includes planning how each structural element will be formatted in various contexts, it also must account for user interface aspects. Unfortunately, this process is frequently slighted in markup projects. As a result, SGML itself has acquired an undeserved reputation in some circles for being complex and difficult to use. In fact, poorly designed markup applications are often at fault.

Many things can be done to make markup easier and more appealing to the user. For example, FrameMaker menus can be customized, both by adding new application-specific commands defined through the FDK and by simplifying the menus in removing access to unneeded capabilities. In this part of the design phase, the project team must consider questions such as the following:

- What is the expected sequence of tasks that users will perform?
- Are there repeated steps that can be automated through the FDK?
- Will users work with complete documents or with fragments?
- Will all documents use the same formatting templates?

Implementing the Application Once the technical goals of the project have been determined, implementation begins. "Technical Steps in FrameMaker Application Development" on page 529 describes the steps involved in implementing a FrameMaker application.

Testing Although writing a DTD or a formatting specification does not use a traditional programming language, it is essentially a programming process. Therefore, even when there is no need for FDK clients, creating an markup application (for FrameMaker or any other tool) is a software development effort and, as such, requires testing. Two types of tests are needed: realistic tests of actual documents and tests of artificial documents constructed specifically to check as much of the application as possible.

All necessary processes must be tested: formatting, markup import, and markup export. Test documents must include FrameMaker documents to save as markup, and markup documents to open in FrameMaker. Testing must also include any processing of legacy documents, including the use of the FrameMaker conversion utility. Finally, once the application is used in production, continued testing should incorporate some actual production documents.

Training and Support Despite the intuitive nature of structured documents, end users cannot be expected to understand the intended use of different element types and attributes simply by working with them. They must be provided with training in the form of documentation, classes, or sample documents. Application-specific training can be combined with training on FrameMaker and other tools to be used in the project.

Provision must also be made for ongoing support. As they use the application, end users will occasionally have questions or encounter bugs. They may need to use structures that have not already been defined.

Maintenance While the application development effort decreases over time, some maintenance should always be expected. In addition to needing bugs fixed, applications may be changed to accommodate new formatting, to encompass additional documents, and to track updates to the DTD.

The Implementation Team So far, the different phases in the development of an markup application have been enumerated. Implementing those phases requires a team of people with different areas of expertise.

All large projects begin with an analysis phase. Participants must include both end users and developers, who will work on the eventual implementation. For an XML or SGML project, the end users include.

- Authors, editors, and graphic designers (production formatters) who will use the editing and publishing application
- Subject-matter experts familiar with the content requirements of the documents to be processed.

Developers include individuals with expertise in

- Markup
- FrameMaker
- Any other markup tools to be used
- Additional software tools, such as document management systems and database packages

Additional participants in the analysis may contribute significantly to the project definition even if they will neither develop nor use the completed application. For example, the organization's Web advocate may be able to identify some requirements that do not affect the rest of the team.

The skill sets required within the implementation team include document design, markup knowledge, setting up FrameMaker formatting templates, and setting up formatting rules that control automatic application of the desired graphic design to structured documents. If the FDK is used, programming skills are also needed. Of course, technical writing skills in documenting the finished application are a valuable contribution to the completed effort.

Technical Steps in FrameMaker Application Development

This section enumerates the tasks involved in developing a FrameMaker application and defines the files containing modules of the application that the application developer maintains.

Element Definition Documents

At the heart of every FrameMaker application is an element definition document (*EDD*). An EDD provides three types of information:

- The definitions of the elements that can occur in a document, including their allowable content and attributes. These definitions can be automatically extracted from the DTD, if there is one.
- The formatting rules that define the visual characteristics of various document components.
- Other information governing behavior of elements, including rules for automatically inserting several elements in response to a single user action, preparing for the use of document fragments, and so forth.

Since an EDD is a structured document, the FrameMaker Guided Editing feature assists developers in creating proper EDDs. In particular, developers need not remember where to insert element definitions, context specifications, or format rules. The Structure View and Element Catalog guide them in creating these constructs correctly. The formatting of the EDD increases its readability: Different fields are clearly labeled, and spacing, indentation, and different fonts emphasize its organization. Developers can increase the accessibility of information by grouping element definitions into sections and explaining them with extensive comments. These characteristics are illustrated by the following fragment:

Element Definition Document (EDD) for Reports

This Element Definition Document defines the structure rules for a report.

Element (Container): Report

Valid as the highest-level element.

General rule: (Title, Abstract, Contents, Chapter+, Appendix*)

Report and Chapter Structure

A report consists of Chapters and Appendices, preceded by a Title, Abstract, and Table of Contents. The Chapters and Appendices may be divided into Sections. Each Chapter, Appendix, and Section has a Head (or title).

Title of the entire report (the Head element is used for Chapter, Appendix, and Section titles).

Element (Container): Title

Valid as the highest-level element.

General rule: (<TEXT>)

Text format rules

1. **In all contexts.**

Basic properties

Alignment: Center

Line spacing

Height: 12pt

Line spacing is fixed.

Default font properties

Weight: Bold

Size: 36pt

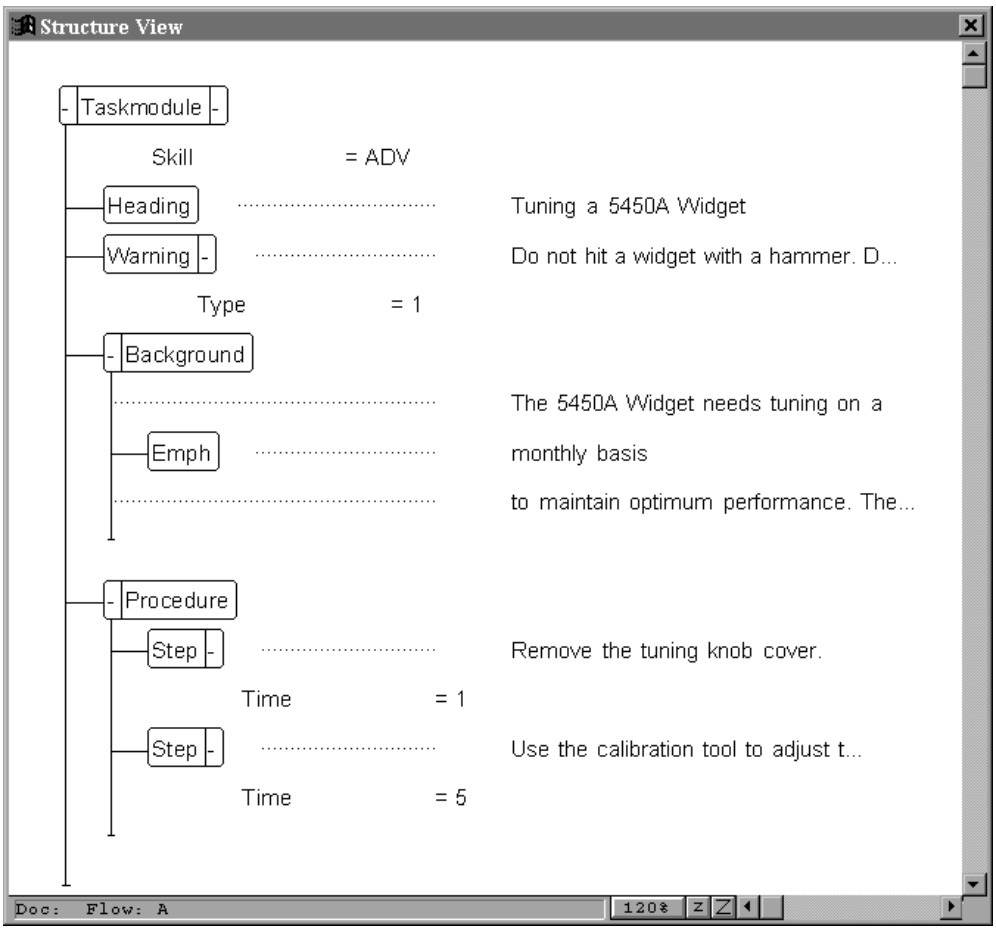
Structured Templates

End users do not use EDDs directly. Instead, the definitions made in an EDD are extracted and stored in a template. A template is simply a FrameMaker document used as a starting point for creating other documents. Developers provide end users with a template that incorporates the structure and format rules from a particular EDD. The template also includes page layout information and formatting catalogs that define paragraph, character, cross-reference, and table styles. It may also contain sample text.

In some applications, information may be formatted in several ways. For example, the same text may need to be formatted for books with two page sizes. FrameMaker can accommodate such requirements. If an EDD's format rules refer to paragraph and character styles in the template's catalogs, new catalogs can be imported from another template to change the document's appearance. New page layouts, table styles, and cross-reference styles can also be imported. If the format rules incorporate specific formatting parameters, the appearance can be changed by importing a new EDD with different format rules.

Moving Data Between markup and FrameMaker

FrameMaker maintains the native element and attribute structure of markup in a *WYSIWYG* environment. The FrameMaker user can easily inspect this element structure in the Structure View. For the maintenance procedure example, the Structure View appears as follows:



Since both the FrameMaker rendition and the markup text file include content and element structure, data can move between the two forms automatically. The user interface is straightforward. The FrameMaker File > Open command recognizes XML and SGML documents. When a user opens one, FrameMaker automatically converts the document instance to a structured *WYSIWYG* document and applies the format rules of the appropriate markup application. To write a structured *WYSIWYG* document to markup, the user simply specifies markup when executing the File > Save As command.

Just as the form of a structured document parallels the form of an markup document instance, an EDD parallels a DTD. FrameMaker can also move definitions of possible

structures between the two forms. Thus, if a new project is based on an existing SGML DTD, FrameMaker can automatically create an EDD from the DTD. For example, given DTD declarations such as

```
<!ELEMENT step - - (#PCDATA)>
<!ATTLIST step time NUMBER #IMPLIED
```

FrameMaker builds the following element definition:

Element (Container): Step
General rule: <TEXT>
Attribute list
1. **Name:** Time **Integer** **Optional**

Since there is no formatting information in the DTD, no format rules are included in the automatically generated EDD. The developer can manually edit this information into the EDD.

Adding format rules to an EDD needs to be done only once. It is common for a DTD to undergo several revisions during its life cycle. When the developer receives an updated DTD, FrameMaker can automatically update an existing EDD to reflect the revision. While the update process preserves existing formatting information and comments in the EDD, it incorporates changes to content models and attribute definitions as well as inserting new element types and removing discarded ones. Furthermore, it generates a short report summarizing the changes so that the application developer may review them.

If a project is not founded on an established DTD, the developer may start implementation by creating an EDD. Once the EDD is finished, FrameMaker can automatically create the corresponding DTD.

Customizing markup Import/Export

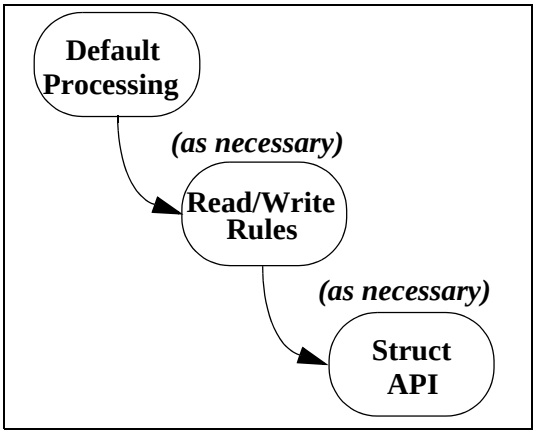
As mentioned, the developer may wish to customize the correspondence between a document's markup and FrameMaker representations for several reasons. Possibilities include:

- Basing the FrameMaker application on an authoring version of an interchange DTD
- Integrating FrameMaker with database or document management tools
- Calculating portions of a document or displaying predefined text when certain elements or attribute values occur or particular entities are used
- Interpreting certain elements, such as graphics or tables, as special objects

To support such requirements, FrameMaker offers three strategies for specifying how abstract markup structures are represented in the *WYSIWYG* publishing environment. Most projects combine two or more of these approaches:

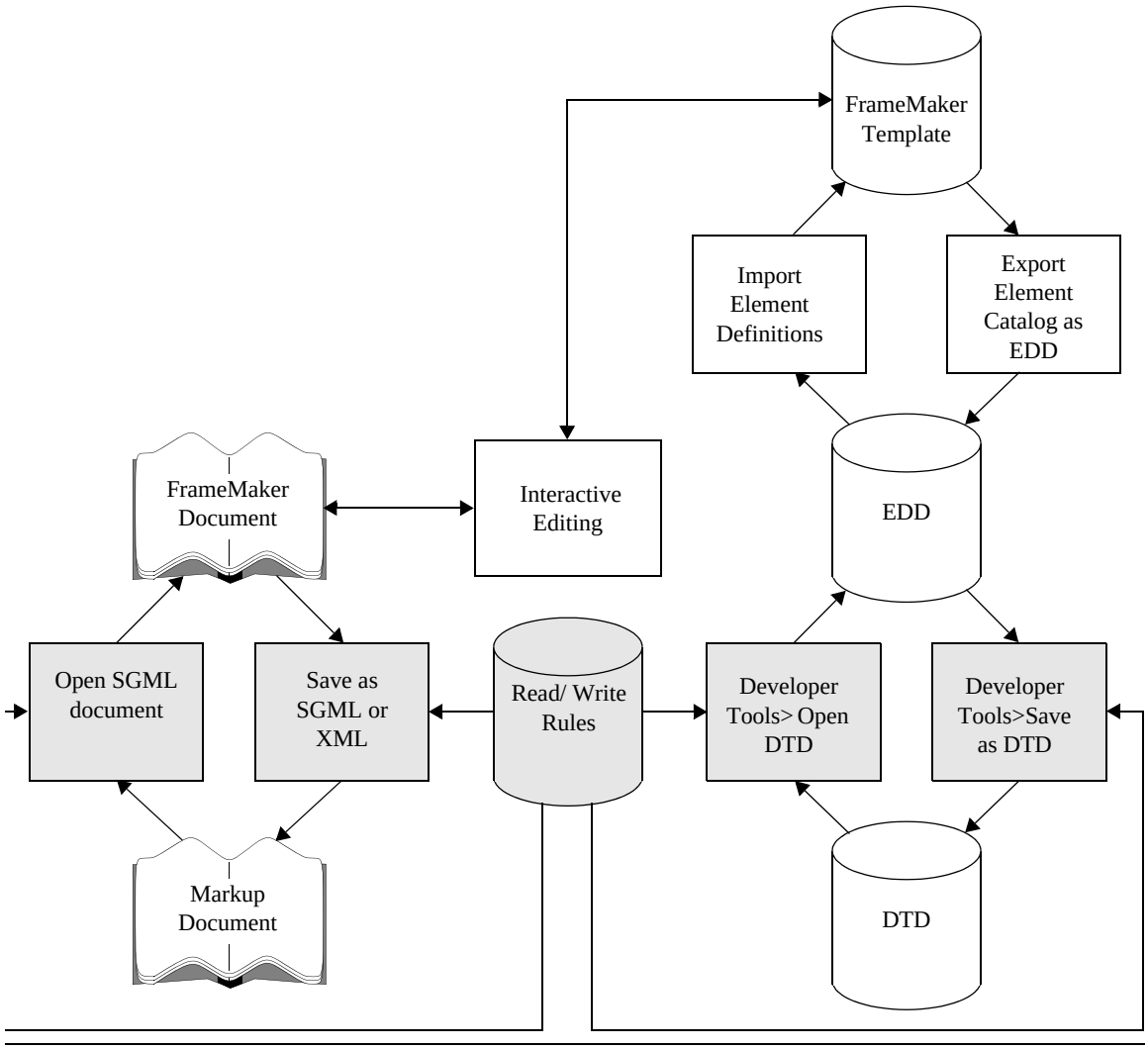
- By default, FrameMaker automatically translates an markup element into a FrameMaker element with the same name (and translates a FrameMaker element to a markup element with the same name); similar processing occurs for attributes. For many elements, the default processing is adequate.
- Some customizing can be accomplished through *read/write rules*. FrameMaker provides read/write rules for customizing that is common to many applications. For example, there are read/write rules for mapping terse NAMELEN-bounded SGML names to longer FrameMaker equivalents; for mapping SGML elements to special-purpose FrameMaker elements such as tables, graphics, or markers; and for associating SDATA entities with special characters. The developer enters read/write rules in a separate file.
- For cases in which read/write rules are insufficient, the FDK includes a C function library called the Structure API, which enables more extensive customizing. The Structure API can be used, for instance, to extract part of a document's text from a database using attribute values as database keys.

These three approaches are diagrammed following diagram:



Read/write rules can be used with all four possible conversions—markup document to FrameMaker document, FrameMaker document to markup document, DTD to EDD, and EDD to DTD. These conversions are shaded in the following diagram, which also shows the FrameMaker commands that invoke them and their interaction with other data files.

Note that in the diagram there is no designated starting point or direction for the flow of information; different projects enter the flow at different stages and follow the arrows in the appropriate direction.



Structure application files

As explained in the preceding section, moving documents and DTDs between their markup and FrameMaker forms involves a number of files. The developer bundles the names of the files needed for a particular application—as well as other information—into an *application definition*. Definitions for all markup applications available on a particular computer system are grouped in an application file. One of the FrameMaker configuration files, `structapps.fm`, is a structure application file that FrameMaker reads whenever it starts. The following example is a fragment of a typical structure application file:

Application Definition Version 5.0

Application name: Maintenance
- **DTD:** task.dtd
- **Read/write rules:** task.rw
- **SGML declaration:** namelen.big
- **DOCTYPE:** taskmodule

Application name: Report
- **Template:** report.tpl
- **Entity locations**
  - **Entity catalog file:** catalog

Since FrameMaker automatically reads `structapps.fm`, end users do not need to know about the structure application file. To import or export markup documents, however, they may need to know the names of available applications. But because an application can be stored in a FrameMaker template or automatically selected from the document type name of an markup document, sometimes even knowing the names is not always necessary. For example, given the preceding structure application file example, when reading an SGML document that begins

```
<!DOCTYPE taskmodule SYSTEM "task.dtd">
```

FrameMaker uses the Maintenance application. Since that application specifies read/write rules, the indicated read/write rules control the import. Furthermore, the Maintenance application is stored in the resulting structured document so that the same read/write rules will be used if the document is later exported back to markup.

The information in an application definition can include:

- The name of a file containing an SGML declaration to be used for export when no SGML declaration appears at the beginning of imported SGML documents.
- The name of a template to be used to create new documents during import. The template incorporates the element definitions imported from an EDD as well as page layout information and formatting catalogs. A template is needed only for applications that include markup import.
- The name of the file containing the DTD to be used for export. This field is not needed for applications that only require import, since imported markup documents always include a document type declaration. When it does appear, the “DTD for export” field actually contains the name of a file containing DTD declarations (sometimes called an external DTD subset) suitable for use in document type declarations such as the one shown in the preceding example for the Maintenance application.

- One or more document type names that trigger use of the application on import.
- Identification of a structure API client to use during import or export.
- The name of the file containing the application's read/write rules.
- Files containing external entities (identified by entity name or public identifier).
- Search paths for subfiles of read/write rules and external entities.
- An entity catalog

Legacy Documents

In general, the conversion of unstructured documents to structured documents (or to markup documents) cannot be completely automated. FrameMaker offers a utility that adds structure to unstructured FrameMaker documents. This is done by mapping paragraph and character styles as well as other objects—such as footnotes, cross-references, markers, and tables—to elements. Some manual polishing of the resulting documents is expected; the amount of editing depends on the discipline with which formatting catalogs were used in the original and on whether a unique set of elements corresponds to each formatting tag. Since FrameMaker can read several common word processing formats, this utility can be used to structure word processor documents as well. Once any errors in the resulting structured document have been repaired, it can be exported to markup. Thus, FrameMaker provides a path for converting word processor documents to markup.

The rules for mapping between styles and elements are specified in a conversion table. Techniques for defining the mapping are discussed below. Consider a very simple unstructured document that consists of a paragraph tagged `MainTitle` followed by a paragraph tagged `ChapterTitle` and one or more paragraphs tagged `Body`. Each sequence of `Body` paragraphs may be followed by another `ChapterTitle` and more `Body` paragraphs. The `Body` paragraphs contain plain text and may also have some characters with the character format `Emphasis`. A conversion table for such a document appears below:

Wrap this object or objects	In this element	With this qualifier
P:MainTitle	Title	Main
P:ChapterTitle	Title	Chapter
P:Body	Para	
C:Emphasis	EmphPhrase	
E:Title[Chapter], E:Para+	Chapter	
E:Title[Main], E:Chapter+	Doc	

Each row in the table is a separate rule. The first row maps the paragraph tagged `MainTitle` to an element called `Title` and, during the structuring process, associates the qualifier `Main` with this particular `Title` element. The next row specifies that each `ChapterTitle` paragraph is also a `Title`, but qualifies these `Titles` by `Chapter`. The next row says that `Body` paragraphs become `Para` elements, and the following row wraps text ranges with the character format `Emphasis` to `EmphPhrase` elements.

The rules in the next two rows combine these low-level elements to define higher levels in the hierarchy. The penultimate row says that a `Title` element followed by one or more `Para` elements is a `Chapter`, provided that the `Title` is qualified by `Chapter`—in other words, a `Chapter` is a chapter `Title` followed by some `Paras`. The last row says that a `Title` qualified by `Main` followed by some `Chapters` is a `Doc`.

Note that there is no need to specify that an `EmphPhrase` element can appear within a `Para`. FrameMaker automatically places text range elements within the element for the paragraph in which they appear.

Application Delivery

Once an application has been completed, it must be delivered to the end users. The developer can either install the completed application on the end users' system or network, or provide instructions for doing so. At a minimum, end users need copies of all templates and FDK clients. The FDK clients must be installed according to platform-dependent conventions. If end users will be structuring existing unstructured documents, they will also need copies of all conversion tables used. Finally, if end users will be exporting or importing markup documents, their structure application files must be modified for the new application. They need access to the read/write rules, entity catalog, DTD for export, and (for SGML structure applications) an SGML declaration.

Typical Application Development Scenarios

Although creation of each component of a FrameMaker application is straightforward, novice developers may need some guidance as to the order in which to create these components. Suggestions are especially appropriate, because the strategy for creating an application can vary considerably with the requirements of a particular project. This section therefore describes three scenarios and how developers may proceed. The scenarios begin after the analysis and design phases have been completed. In all three, the developer implements a few elements at a time and stops to test the growing application frequently.

Starting from Existing SGML Documents

In the first scenario, there is an existing DTD and sample markup documents. The developer selects a short sample document and starts development by supporting the element types that occur in it. Later, the developer adds more complicated documents. Application development proceeds as follows:

1. The developer creates a new file of read/write rules and enters rules that reflect any decisions made during the analysis phase of the project. In particular, the developer enters rules to

- Identify elements used for graphics, tables and table components, and cross-references
- Discard elements and attributes that the organization will not use even if they are defined in the DTD
- Rename elements and attributes to reflect the organization's terminology
- Replace terse, abbreviated markup names with longer, more readable FrameMaker versions
- Specify FrameMaker names with multiple capital letters. For instance, the developer may map the element generic identifier `BLOCKQUOTE` to the FrameMaker element tag `BlockQuote`. Such rules will not be needed with SGML applications if the SGML declaration does not force general names to uppercase (that is, if case is significant in the SGML names).

2. The developer defines a new markup application in the structure application file that invokes the read/write rules. If appropriate, the application definition specifies an XML or SGML application and maps any public identifiers used in the DTD to system filenames.
3. Using the new application, the developer creates an EDD from the DTD with the File > Structure Tools > Open DTD command. The resulting element declarations may be grouped into sections, and some added comments in the process.
4. The developer opens the selected sample document. The resulting FrameMaker document serves as the basis of a template. In this document, the developer defines page layouts, specifying the number of columns on the page, the width of any side-heads, page numbers, and so forth.
5. The application developer returns to the EDD and begins to define formatting rules. In general, formatting specifications are unnecessary for higher level elements, such as those for chapters and sections. The application developer concentrates on elements that contain text and their immediate ancestors, such as elements for list items, notes, cautions, emphasized phrases, and cited titles.
6. After providing format rules for a few elements, the application developer tests them by using the File > Import > Element Definitions command to import the element definitions from the EDD into the sample document. This command reports any formatting catalog entries mentioned in format rules but not defined in the sample FrameMaker document. The developer defines all reported formats and glances through the sample document to verify that the formatted elements appear as intended.

   The developer then returns to the EDD, correcting any errors in the format rules and adding some more before testing them again, until the sample document is completely formatted. At that point the developer tests some other documents. After creating a base for further development, the developer continues systematically through the DTD, making and testing context-sensitive format rules for all necessary elements.
7. The developer may add more read/write rules. If the developer changes additional element tags, the original names must be replaced by the new versions throughout the

EDD, in the definitions of the changed elements as well as in definitions that refer to those elements. To update the names in the EDD, the developer uses the File > Developer Tools > Import DTD command, which reprocesses the DTD according to the current read/write rules and updates the existing EDD. It writes a report listing the changes made.

8. To create a template from the sample document, the developer deletes all content and saves the result.
9. The developer can then use the template to test the editing environment by adding automatic insertion rules to the EDD, specifying, for example, that when the end user inserts a `List` element, FrameMaker should automatically insert an `Item` element within it. The developer also considers whether an FDK client can provide input accelerators, such as automating frequent sequences of editing steps. The developer may also customize the FrameMaker menus for end users, perhaps removing developer-oriented commands or (to prevent the end user from overriding automatic format rules) removing formatting commands.
10. For markup import, the developer adds the template to the application definition in the structure application file, and adds read/write rules for special characters and other entities. Once again, the developer opens the markup version of the sample document, tests the result, and modifies the application until the results are correct.
11. If read/write rules are insufficient to import any elements, the developer uses the structure API to develop a client with the requisite functionality.
12. If markup export is also required, the developer tests it, adding other read/write rules or structure API functions, if necessary.
13. Once formatting, import, export, and editing functions have been tested, the developer delivers the completed application, along with documentation, to the end user.

Building a New Application

Another typical use of FrameMaker involves developing the document type definition from scratch. In this scenario, FrameMaker is used to write documents that are then exported to markup. Again, development work relies on a thorough data analysis.

1. Instead of starting with a DTD, the developer begins to develop an EDD. Guided Editing makes the mechanics of editing the EDD simple. It is the developer's personal preference whether to define both content rules and attribute definitions for one element at a time, or to complete all content rules before defining any attributes. The developer may also define content rules for numerous elements before adding any format rules.
2. The developer creates a template and then uses the File > Import Element Definitions command to interpret the EDD and store the resulting element catalog in the template.
3. The developer uses the template to test sample documents. As in the scenario of the previous section, the developer may find it easier to keep track of the work by repeating the edit-import-test cycle a few elements at a time.

4. When the EDD is complete, the developer uses the File > Structure Tools > Save As DTD command to create an equivalent DTD.
5. The developer creates a structure application that uses the generated DTD for export.
6. The developer tests the export of the sample document and adds read/write rules and a structure API client as necessary. If the developer does not want to use the FrameMaker default representation for special object elements, editing the corresponding element and attribute definition list declarations is the thing to do, in addition to providing the associated rules.
7. The developer may edit the generated DTD by allowing for tag omission and providing appropriate short reference mapping, short reference use, and entity declarations. Since FrameMaker exports all comments in the EDD to the generated DTD, the developer reviews comments carefully and retains comments pertaining to the structures being defined, but discards comments specific to FrameMaker formatting, since they do not necessarily apply to all uses of the DTD.
8. If the application requires markup import as well as export, the developer tests conversion in the import direction as well, modifying the application to correct any errors.
9. Finally, the developer delivers the application to end users and prepares for the project's maintenance phase.

Working With Legacy Documents

The final scenario assumes there are existing unstructured FrameMaker documents. The developer can take either of two general approaches: constructing a conversion table to structure the legacy documents and using the resulting structure to start an EDD; or creating an EDD based on an analysis of the legacy documents and then creating a conversion table to reflect the element definitions in the EDD. The following steps outline the first approach:

1. The developer selects a typical unstructured document and uses the File > Structure Tools > Generate Conversion Table command to create a new conversion table based on the format tags of the unstructured document. This command puts a row in the table for each type of FrameMaker object and format tag used in the sample document. The initial conversion table might appear as follows:

Wrap this object or objects	In this element	With this qualifier
P:MainTitle	MainTitle	
ChapterTitle	P:ChapterTitle	
P:Body	Body	
C:Emphasis	Emphasis	
G:	GRAPHIC	
F:flow	FOOTNOTE	
T:Table	Table	

2. The developer edits the conversion table, changing entries in the second column, adding entries in the third column, and adding new rows for higher level structures.
3. The developer uses the conversion table to add structure to the sample document, ignoring the lack of formatting in the resulting structured document but correcting as many structural errors as possible by editing the conversion table and reapplying it.
4. With the File > Structure Tools > Export Element Catalog as EDD command, the developer extracts an EDD that has an element definition for each element type that appears in the sample document. Although these element definitions define the content of the elements very generically—each is allowed to contain text as well as any defined element—the EDD provides a skeleton that the developer can further develop.
5. The exported EDD lists the element definitions alphabetically, according to their tags. The developer can rearrange them into a logical order if desired—easily moving them around using drag-and-drop in the Structure View—and then replace the content rules with more restrictive ones and add format rules.
6. The sample document (and other automatically structured documents) can be reformatted by using the File > Import Element Definitions command from the EDD into the generated structured document
7. The developer then continues to finish the application, as in the previous scenario. However, since page layouts and formatting catalogs were defined in the original unstructured document, the template development effort does not need to be repeated.

Conclusions

This paper has described how all general-purpose markup systems require application-specific information. It has enumerated the robust and flexible set of tools that FrameMaker provides for creating markup editing and publishing applications. Application development can be divided into separate steps, each of which is straightforward. The beginning of “Overview of FrameMaker Application Development” on page 521 listed the major parts of

an application. The following table reviews the list, along with the application components in which the information is specified:

Application Component	File
Element and attribute definitions	Edd (can be derived automatically from DTD)
Format rules	EDD
Page layouts	Template
Markup representation of elements and attributes	Read/write rules and structure API client

The structure application file bundles this information in the table so that it can be easily accessed. Application development involves creating these files along with data analysis, documentation, and maintenance activities.

Adobe, the Adobe logo, and FrameMaker are trademarks of Adobe Systems Incorporated.

©1997 Adobe Systems Incorporated. All rights reserved. Printed in the USA. 000000 9/97.

Glossary

This glossary contains common terms used by FrameMaker, XML, and SGML. For references to more information about the terms, see the index.

ancestor An element that contains a given element in a document's structure. For example, if a `Section` element contains a `Head` element followed by a `Paragraph` element, and the `Paragraph` contains a `Variable` element, the `Paragraph` and `Section` elements are both ancestors of the `Variable` element, but the `Head` element is not an ancestor of the `Variable` element. *See also* descendant, child element, parent element, and sibling.

API Application Programming Interface. Enables developers to create API clients with other applications, such as databases, document management systems, CAD tools, and user interfaces, for automation, database publishing, HTML conversion and other purposes.

application definition A data structure (and the associated files) describing part of a complete XML or SGML application assembled with FrameMaker. You store application definitions in the `structapps.fm` file.

attribute A place to supply information about an element other than its hierarchical position and structure. An attribute value does not add content to a document.

attribute definition The construct used to define a single attribute in a FrameMaker EDD or a DTD.

attribute definition list declaration In markup, the declaration that provides the list of attribute definitions for one or more elements. Also called an ATTLIST. *See also* element declaration.

book A grouping of FrameMaker documents that lets you work with them as a single unit. Lets you generate a single table of contents or other file from the documents, and simplifies printing, numbering, cross-referencing, and formatting.

CALS Continuous Acquisition and Life Cycle Support. The US Department of Defense standard for the electronic delivery of documents.

catalog A floating palette that stores predefined paragraph, character, or table formats.

CDATA In markup, character data. In character data, no markup is recognized, other than the delimiters that end the character data. *See also* NDATA, #PCDATA, RCDATA, and SDATA.

child element An element that is contained in a given element and that is one level below the given element. For example, if a `Section` element contains a `Head`

element followed by a Paragraph element, and the Paragraph element contains a Variable element, the Head and Paragraph elements are both child elements of the Section element, but the Variable element is not. *See also* parent element, ancestor, descendant, and sibling.

concrete syntax In SGML, a set of choices on the markup a document will use. Since SGML does not require any particular values for these choices, an SGML document requires a concrete syntax so a parser can correctly interpret it. *See also* reference concrete syntax.

container element In FrameMaker, an element that can contain text, other elements, or both. Contrasts with certain specific element types—for example, a cross-reference element, which can contain nothing other than the cross-reference.

content model In markup, the part of an element declaration that specifies both a model group and exceptions that define the allowed content of the element. Each markup element declaration has either a content model or declared content. *See also* content rules, declared content, general rule, and model group.

content rules In FrameMaker, the part of an element declaration that specifies both the element's type and the kind of contents the element can have. *See also* format rules, content model, and general rule.

conversion table In FrameMaker, a table associating parts of an unstructured document with their structured counterparts, used in converting an unstructured document to a structured document.

cross-reference A passage in one place in a document that refers to another place, its cross-reference source, in the same or a different document.

cross-reference source The place referred to by a cross-reference.

data In markup, the characters of a document that represent the inherent information content. Such characters are not recognized as markup. *See also* markup.

data content notation In markup, an application-specific interpretation of an element's data content, or of a data entity, that usually extends or differs from the normal meaning of the document character set. Frequently used to identify the format of an external entity containing a graphic.

declaration In markup, markup that controls how other markup of a document is to be interpreted.

declared content In an markup element declaration, specifies that the defined element's content is one of the reserved types CDATA, RCDATA, or EMPTY.

declared value In an markup attribute definition, determines the type of attribute value, such as ID or NUTOKEN, that is valid when the attribute is specified. Although

	markup does not define the term *attribute type*, you can loosely think of an attribute's declared value as its type.
default value	In markup, the portion of an attribute definition that indicates whether an attribute is required and what value to use if the user doesn't specify one. In FrameMaker, refers only to the value to use if a user doesn't supply a value for an attribute.
delimiter	In markup, a character string used to identify a piece of markup or to distinguish markup from data. For example, > (greater-than sign) is the default closing delimiter for element tags.
descendant	Any element that is below a given element in a document's structure. For example, if a `Section` element contains a `Head` element followed by a `Paragraph` element, and the `Paragraph` element contains a `Variable` element, the `Variable` element is a descendant of both the `Paragraph` and the `Section` elements, but not of the `Head` element. *See also* ancestor, child element, parent element, and sibling.
DOCTYPE	In markup, the reserved name that follows the opening delimiter of a DTD. Informally used to refer to the document element.
document	A collection of information that is processed as a unit. A FrameMaker document is any file in FrameMaker format. A markup document includes an SGML declaration (for SGML), prologue, and document instance set.
document element	In markup, the highest-level element in a document. The generic identifier of this element is specified immediately after the `DOCTYPE` reserved name in the DTD.
document instance	In markup, the portion of a document that contains markup and data for a particular project such as a memo or book.
document type	A class of documents having similar characteristics, such as technical manual or internal memo.
document type declaration	In markup, a document type declaration (DTD) associates a document element with a set of declarations (the document type declaration subset).
document type declaration subset	In markup, a set of declarations determining such things as the markup to allow in a document and the elements and attributes for a document set. *See also* external DTD subset and internal DTD subset.
DTD	*See* document type declaration subset.
EDD	*See* element definition document.
element	A structural unit of a document. Holds and organizes the contents of the document.

Element Catalog	In FrameMaker, the information extracted from an EDD and stored within each structured FrameMaker document. Makes an external element definition document unnecessary. *See also* element definition document.
element declaration	In markup, information describing a particular element. Includes both a name (generic identifier) for the element and content rules. A markup document has an element declaration for each allowed element.
element definition	In FrameMaker, a set of rules describing an element. Includes a name (tag) for the element, content rules, and (optionally) context-sensitive format rules. A structured document has an element definition for each element allowed. *See also* content rules and format rules.
element definition document	A FrameMaker document that contains a set of element definitions for a class of documents. Can also include information on system defaults and on a structure application with which to associate this information. Also called an EDD.
element tag	In FrameMaker, the name assigned to an element and stored in the Element Catalog. *See also* generic identifier.
EMPTY	Keyword in an element definition indicating that the element cannot have content. In markup, `EMPTY` is a declared content.
end-tag	In markup, the markup that indicates the end of an element.
entity	In markup, a collection of characters that can be referenced as a unit. Used for many purposes in markup, such as graphics or frequently used sets of characters.
exclusion	An exception to the general rule or content model of an element. Specifies other elements that cannot appear anywhere in the element or in its descendants. Exclusions are not allowed in XML.
external cross-reference	In FrameMaker, a cross-reference to a source in a different file. Markup does not define this concept.
external DTD subset	In markup, an informal term for an external entity for which an external identifier appears at the beginning of a document type declaration and that is automatically referenced at the end of the document type declaration subset.
external entity	In markup, an entity that specifies an external object, such as a file.
facet	A pictorial representation of graphical data.
FDK client	In FrameMaker, any application created using the Frame Developer's Kit. *See also* Structure API client.
flow	*See* , "text flow."

format rules In FrameMaker, the part of an element definition that specifies which predefined format to apply to an element. Format rules can use different formats for different contexts in a document. *See also* content rules.

general entity In markup, an entity that can be referenced from within the content of an element or an attribute value literal.

general rule In FrameMaker, a rule that specifies valid contents for an element and the order in which the contents can appear. Equivalent to the declared content of an element or the model group part of the content model of an element in markup. *See also* content rules.

generic identifier In markup, the name identifying an element. *See also* element definition and element tag.

highest-level rule In FrameMaker, a read/write rule that is not a subrule of another read/write rule.

HTML Hypertext Markup Language. An encoding system used to describe the content and organization of an electronic document published on the World Wide Web.

ID attribute An attribute of type `ID`, frequently used as an identifier to mark the source of a cross-reference. In a single document, a particular value for an `ID` attribute can be used only once.

IDREF attribute An attribute whose value must be that of an `ID` attribute in the same markup document or FrameMaker document or book. Frequently used for cross-references.

impliable attribute In markup, an attribute whose value does not have to be supplied. If a document doesn't supply a value, it is up to the processing software to correctly interpret the attribute. Such attributes use the default value `#IMPLIED`.

inclusion An exception to the general rule or content model of an element. Specifies other elements that can appear anywhere in the element or in its descendants.

invalid element An element with contents that do not conform to content rules. May be missing required child elements, may not have a definition in the EDD or DTD, or may have text or child elements in a position not allowed by its content rules or by the exclusion and inclusion rules of its ancestors.

internal cross-reference In FrameMaker, a cross-reference to a source in the same file.

internal DTD subset In markup, an informal term for the declarations in a document type declaration that occur within brackets (dso and dsc delimiters) in the markup document entity, rather than being in an external entity.

internal entity In markup, an entity whose replacement text is determined solely by information in its declaration.

ISO public entity In SGML, an entity that occurs in one of the entity sets defined in Annex D of the SGML Standard. These entities provide commonly used special characters.

marker In FrameMaker, a nonprinting character an end user inserts (such as an index entry) to indicate various types of information.

markup Text added to the data of a document in order to convey information about it, such as hierarchical structure or formatting. This document also uses *markup* to generally refer to XML and SGML.

markup minimization In SGML, any of various conventions for omitting markup in a document, including shortening or omitting tags and shortening entity references.

model group In markup, an ordered list that specifies valid contents for an element (such as child elements) and the order in which the contents can appear. A model group is similar to a FrameMaker general rule.

NAMECASE parameter In SGML, the part of the SGML declaration that determines case-sensitivity of markup.

NDATA In SGML (and implicitly XML), non-SGML data. `NDATA` is data that needs special processing by the markup application. `NDATA` is typically used, for example, when representing graphics—in XML the graphic data would be non-parsed data. *See also* CDATA, #PCDATA, RCDATA, and SDATA.

parameter entity In markup, an entity that can be referenced only within a DTD.

parent element An element that contains a given element and is one level above it in the hierarchy. For example, if a `Section` element contains a `Head` element followed by a `Paragraph` element, the `Section` element is the parent element of the `Head` and `Paragraph` elements, but not of the `Variable` element. *See also* child element, ancestor, descendant, and sibling.

parser *See* validating parser.

#PCDATA In markup, parsed character data. This is normal text that can include markup to be parsed. Occurs in an markup element's model group and corresponds to <TEXT> in a FrameMaker element's general rule. *See also* CDATA, NDATA, RCDATA, and SDATA.

prefix Text that is automatically placed before the content of an element. In FrameMaker, defined as part of the formatting of an element. For example, a `Quote` text range element might have an open quotation mark as its prefix and a close quotation mark as its suffix. *See also* suffix.

processing instruction In an markup document, a way of indicating that the application needs to perform some special processing. For example, you can use a processing

	instruction to indicate a location in an markup document that should have a page break.
public identifier	In markup, a way of identifying an external entity. Formal public identifiers have a specified syntax that includes an identifier of the owner of the entity and an indication of the markup construct it provides. Formal public identifiers are typically available to any user of markup, not just the users at a particular company. Informal public identifiers may be available more widely than a single document or system, but perhaps no more widely than within a single company. *See also* system identifier
RCDATA	In markup, replaceable character data. In replaceable character data, no markup is recognized, other than character and entity references. RCDATA is valid only in SGML. *See also* CDATA, NDATA, #PCDATA, and SDATA.
read/write rule	In FrameMaker, interpreted commands you supply to modify how the software translates between FrameMaker and markup documents.
reference concrete syntax	In SGML, a particular concrete syntax defined by the SGML standard. *See also* concrete syntax.
reference page	An underlying page that stores repeatedly-used graphics and formatting information.
Rubi text	Small characters that appear above Japanese-language characters to indicate pronunciation.
rule	*See* SGML read/write rule.
SDATA	In SGML, specific character data. One common use is for specific characters that might not be in the standard character set. *See also* CDATA, NDATA, #PCDATA, and RCDATA.
SGML	An acronym for Standard Generalized Markup Language.
SGML application	Rules that apply SGML to a text processing application. Includes a formal specification of the markup constructs used in the application, expressed in SGML. Can also include non-SGML definitions of semantics, application, conventions, and processing.
SGML declaration	In SGML, the part of a document that tells a parser how to interpret markup in the document.
SGML read/write rule	*See* read/write rule.
sibling	Elements at the same level in the structure and with the same parent element. For example, if a `Section` element contains a `Head` element followed by a `Paragraph` element, the `Head` and `Paragraph` elements are siblings. *See also* ancestor, descendant, child element, and parent element.
source	*See* cross-reference source.

start-tag In markup, the markup that indicates the beginning of an element.

Structure API client In FrameMaker, an FDK client created to change the translation between FrameMaker and markup documents. *See also* FDK client.

subrule In FrameMaker, an read/write rule that is part of another rule.

suffix Text that is automatically placed after the content of an element. In FrameMaker, a prefix is defined as part of the formatting of an element. *See also* prefix.

system identifier In markup, a way of identifying an external entity that's specific to the particular document or system. *See also* public identifier.

template In FrameMaker, a document used to create new documents. A template can include all the formats, structure descriptions, and other information you need to create a document.

<TEXT> In a FrameMaker element's general rule, indicates that the element can directly contain text characters and elements included by itself or its ancestors. <TEXT> corresponds to #PCDATA in a markup element's model group.

Text entity An entity whose replacement text can contain both data and markup.

text flow The text in a series of connected text frames. A text flow can also be contained in a single text frame, not connected to any other frame. A text flow with elements is a structured text flow.

text inset Text imported by reference.

<TEXTONLY> In a FrameMaker element's general rule, indicates that the element can directly contain text characters and cannot contain elements included by an ancestor. By default, on export <TEXTONLY> corresponds to a declared content of RCDATA in an SGML element's definition, or PCDATA in XML. On import FrameMaker translates declared content of RCDATA or CDATA to <TEXTONLY>.

valid document A structured document that conforms to all its content rules. Every element in the document must be valid. In FrameMaker, every structured flow must have a highest-level element that is allowed at the highest level.

valid element An element with contents that conform to its own content rules and to the inclusion and exclusion rules of all of its ancestors.

validating parser In markup, a software module that parses the markup of an XML or SGML document and determines that the document structure conforms to a provided DTD.

variable In FrameMaker, text that is defined once but can be used several times. Similar to some varieties of XML or SGML entity.

XML An acronym for Extensible Markup Language. By definition, XML is a subset of SGML.

Index

A

absolute values, in format rules 128
Advanced properties (text formatting) 154-155
{after} sibling indicator 135, 189
alignment settings
  for paragraphs 148
  for table cells 155
all-contexts rules (for formatting) 133, 187
ampersand (&)
  in conversion tables 478
  in format rules 137
  in general rules 112
ancestors
  describing relationships to siblings 135, 189
  inheriting formats from 127-131
  tags in context rules 134, 188
  tags in level rules 139
anchored frame (rule) 365
anchored frames. *See* graphics
angle brackets (< >)
  in prefix or suffix strings 158
  with attributes in prefix or suffix rules 162
{any} sibling indicator 135, 189
ANY keyword, in general rules 113
application definition files 41, 50-70
  contents of 51
  default information 54
  defining applications in 53
  document elements 55
  DTDs for import and export 56
  editing 51
  entity catalogs 58-59
  external entities 60-61
  filename extensions, specifying 56
  files for rules documents 64
  individual entities 59
  length of log files 70
  namespaces, enabling 56
  public identifiers 61
  read/write rules documents 55
  search path for external entities 63-64
  SGML declarations 57
  structure API clients 66
  templates for import 57
Asian text spacing properties (text formatting) 155
asterisk (*)
  in conversion tables 478
  in format rules 135, 188
  in general rules 112
attribute definitions 169-183
  attribute name 172
  attribute type 172
  correspondence to markup languages 229-231
  default values 176
  hidden and read-only specification 174
  list of choice values 175
  optional-value specification 173
  range of numeric values 175
  required-value specification 173
attribute names

correspondence to markup languages 231
renaming for markup languages 235
restrictions on 172
attribute (rule) 367
attribute types
changing for markup languages 243
correspondence to markup languages 230
list of possible 172
attribute values
default 176
default values for markup languages 242
for choice attributes 175
for IDReference attributes 180
for numeric attributes 175
for UniqueID attributes 178
renaming for markup languages 235
required or optional 173
attributes
comparison with markup languages 16
for a prefix or suffix 162, 182
for cross-referencing 176-180
for formatting objects 189
for formatting text 136-137
for formatting text or objects 180-182
for identifying overrides 482
hidden 174
how end users work with 170
in conversion tables 479
read-only 174
uses for 169
writing definitions for 171-176
attributes, default translation to and from markup languages
attribute definitions 229-231
attribute names 231
attribute types and declared values 230
attributes, modifying translation to and from markup languages
changing attribute types 243
discarding attributes 241
renaming attributes 235
renaming values 235
specifying default values 242
specifying read-only 244
attributes, read/write rules for 355
attribute 367
drop 373
fm attribute 396
fm element 397
implied value is 406
is fm attribute 414
is fm property 424
is fm property value 426
is fm value 439
value 464
autoinsertion rules 119-120
autonumbers
defining formats for 153
when to use 162

B

Basic properties (text formatting) 147-150
batch import and export (UNIX) 517-520
{before} sibling indicator 135, 189
{between} sibling indicator 135, 189
books
comparison with markup languages 19
inheritance of text formats in 131
internal and external references 322
vs. text entities in markup languages 347
books, default translation to and from markup languages

on export to markup languages 350-351
on import from markup languages 348-349
books, modifying translation to and from markup languages
identifying book components 351-353
suppressing creation of PIs 353
books, read/write rules for 356
generate book 403
output book processing instructions 447
put element 403
use processing instructions 403
brackets ([ ])
in cross-reference formats 44
in format rules 136, 190

C

CALS table model 487-492
attribute structure 491
colspec elements 490, 491
element and attribute declarations 488
element structure 490
spanspec elements 490, 491
CALS tables
EDD object format rules and 185
read/write rules for 493-495
CALS tables, translating from markup languages
CALS attributes for formatting 274, 282
colspec and spanspec elements 274, 283
formatting properties for 276-279
how CALS tables translate 273
caret (^), in error messages 124
CDATA entities, default translation of internal 252
character formats
applying to particular contexts 145
finding errors in 166
wrapping text formatted without 483
character map (rule) 369
character set mapping 501-507
characters allowed
in attribute names 172
in choice attribute values 175
in conversion tables 475
in element tags 92
child elements
exclusions for 118
in general rules 112
inclusions for 117
inserted automatically 119-120
choice attributes 172
formatting elements with 137
specifying values for 175
comma (,)
in conversion tables 478
in general rules 112
commenting an EDD
with comment elements 95, 100
with paragraph elements 90
commenting read/write rules 214
condition settings 342
conditional text, default translation to and from XML 341
on export to XML 343
on import from XML 344
conditional text, modifying translation to and from XML 344
containers
automatic descendants for 119-120
defining elements for 93-97
general rules for 111-116
validity at highest level 116
content rules 111-118

comparison with markup languages 16
debugging 123
overview of 110
translation to markup languages 118
context labels, in format rules 142
context rules (for formatting) 134-138, 188-190
ancestor tags in 134, 188
attributes in 136-138, 189
order of clauses in 137, 190
wildcard characters in 135, 188
conversion tables 471-485
adding rules to 474-481
attributes in 479
building tables from format tags with 484
columns and rows in 471, 474
documents for holding 472
flagging format overrides with 482
format and element tags in 473, 475, 476
generating initial 473
nesting graphics or tables with 483
object type identifiers in 476
order of rules in 472
promoting graphics or tables with 481
qualifiers for element tags in 475, 480
setting up from scratch 474
testing and correcting 484
updating 474
wrapping elements with 477
wrapping objects with 476
wrapping sequences with 478
wrapping untagged text with 483
cross-reference formats
for FrameMaker elements 43-44
cross-references
comparison with markup languages 21
defining elements for 99-102
finding errors in formats of 197
formats for 43-44
IDReference attributes for 179
object format rules for 194
UniqueID attributes for 178-179
cross-references, default translation to and from markup languages
internal and external references 322
on export to markup languages 322
on import from markup languages 323
cross-references, modifying translation to and from markup languages
external cross-references 326
maintaining attribute values 326
renaming format attributes 325
translating elements as references 324
translating elements to markup language text 326
cross-references, read/write rules for 356
fm element unwrap 397
fm property 399
is fm cross-reference element 418
is fm property 424
is fm property value 426
is fm value 439
value is 399
CSS
comparison with EDD format rules 218
differences in translation 222
fmcssattr 222
generating from EDD ??-63, 217-226
generating on command 224
generating on Save As XML 225
managing generation 62
structure application definition 225

D

debugging. *See* errors
default
  attribute values 176
  general rules 115
  initial structure for tables 122
  SGML declaration 497-499
descendants
  exclusions for 118
  inclusions for 117
  inserted automatically 119-120
DOC PI markers for storing entities and PIs 250
DOCTYPE elements 55
document type declarations (DTDs)
  comparison with EDDs 13
  creating from an EDD 106
  EDD content rules and 118
  EDD object format rules and 185
  EDD text format rules and 125
  errors when translating to an EDD 79
  external DTD subsets 14, 41, 77
  saving an EDD as 78
  SGML declarations and 107
  specifying location of 56
  updating an EDD from 79
  See also markup language, translation to and from
documentation, for applications 42
documents, comparison with markup languages 18
drop content (rule) 375
drop (rule) 373
DTD. *See* document type declarations

E

EDD. *See* element definition documents
Element Catalogs
  creating in a template 103-105
  elements in an EDD catalog 81-88
  exporting to an EDD 80
element definition documents (EDDs) 75-107
  adding comments to 90
  comparison with DTDs 13
  creating from a DTD 78
  creating new 80
  errors when translating to a DTD 107
  exporting an Element Catalog to 80
  high-level elements in 81
  list of elements in 82-88
  overview of developing 30-33, 76
  samples to review 107
  saving as a DTD 106
  setting a structure application in 89
  shortcuts for working in 102
  updating from a DTD 79
element definitions
  attribute definitions in 169-183
  basic steps for writing 91-102
  comments in 95, 100
  comparison with markup languages 16
  creating formats when importing 89
  debugging 104
  element tags in 92
  element types in 95, 100
  errors when importing 104
  for containers, tables, and footnotes 93-97
  for object elements 99-102
  for Rubi groups 97-98
  guidelines for writing 92
  importing into a template 104
  object format rules in 185-198
  organizing in sections 90

structure rules in 109-124
text format rules in 125-167
element (rule) 376
element tags
correspondence to markup languages 231
in conversion tables 473, 475, 476
in element definitions 92
in read/write rules 214
renaming for markup languages 234
element types
comparison with markup languages 15
list of possible 95, 100
elements
comparison with markup languages 14-??
discarding when translating to or from markup languages 240
elements, default translation to and from markup languages
element tags 231
exclusions 233
general rules and model groups 228
inclusions 233
line breaks and record ends 233
elements, modifying translation to and from markup languages
converting markup language elements to footnotes 236
converting markup language elements to Rubi groups 237
renaming elements 234
retaining content but not structure 238
elements, modifying translation to and from markup languagesL
retaining structure but not content 239
suppressing display of content 240
elements, read/write rules for all 355
attribute 367
drop 373
drop content 375
element 376
fm element 397
is fm element 419
preserve fm element definition 448, 449
unwrap 461
else clauses, in format rules 134, 188
else/if clauses, in format rules 134, 188
EMPTY keyword, in general rules 113
end vertical straddle (rule) 379
entities 17
external files for 60-61
ISO public 509-515
searching for external files 63
searching for filename patterns 60
specifying location of 59
specifying search path for 63-64
entities, default translation to and from markup languages
external text entities 255
for marking documents in book 350
how used for variables 330
internal text entities 251
on export to markup languages 248
on import from markup languages 251-256
parameter entities 255
entities, default translation to and from markup langugaes
external data entities 254
entities, default translation to and from SGML
internal CDATA entities 252
internal SDATA entities 252
PI entities 256
SUBDOC entities 255

entities, modifying translation to and form SGML
translating SDATA as elements 263
entities, modifying translation to and from markup languages
changing structure and format of insets 266
discarding external data references 267
renaming entities for variables 258
translating characters as entities 268
translating external system entities as insets 264
translating SDATA as characters 260-??
translating SDATA references 258
translating text entities as insets 265
entities, modifying translation to and from SGML
translating SDATA as characters ??-262
translating SDATA as insets 262
translating SDATA as variables 259
entities, read/write rules for 357
drop 373
entity 381
entity name is 383
external data entity reference 391
is fm char 416
is fm reference element 430
is fm variable 441
reformat as plain text 454
reformat using target document catalogs 455
retain source document formatting 456
entity catalogs 42
format of entries in 59
searching for 59
specifying location of 58-59
uses for 58
entity name is (rule) 383
entity (rule) 381
equation (rule) 385
equations
comparison with markup languages 20
defining elements for 99-102
finding errors in sizes of 197
in conversion tables 477
object format rules for 194
equations, read/write rules for 357
entity name is 383
equation 385
export dpi 387
export to file 389
fm property 399
is fm equation element 420
is fm property 424
is fm property value 426
is fm value 439
notation is 445
specify size in 456
value 464
value is 399
equations, translation to and from markup languages. *See* graphics and equations
errors
in imported element definitions 104
in object format rules 197-198
in structure rules 123
in text format rules 166
when creating an EDD from a DTD 79
when saving an EDD as a DTD 107
exclusions
correspondence to SGML 233
in element definitions 118
when used with inclusions 117
export dpi (rule) 387
export to file (rule) 389
exporting to markup languages. *See* markup

languages, translation to and from
external data entity reference (rule) 391
external dtd (rule) 392
external DTD subsets 14, 41, 77

F

facet (rule) 394
files, for applications 49
{first} sibling indicator 135, 189
first format rules 156-158
  how applied 157
  when to use 162
  with autonumbers 157
fm attribute (rule) 396
fm element (rule) 397
fm element unwrap (rule) 397
fm marker (rule) 398
fm property (rule) 399
fm variable (rule) 402
fm version (rule) 403
Font properties (text formatting) 150-152
footnotes
  converting markup language elements to 236
  defining elements for 93-97
  general rules for 111-116
  in conversion tables 477
  inheritance of text formats in 130
footnotes, read/write rules for 358
  is fm footnote element 421
format change lists
  changes that apply from 164
  defining 163
  limits on values in 165
  referring to 144, 145
format overrides, flagging in conversion tables 482
format rule overrides
  object format rules and 187
  text format rules and 127
format rules
  comparison with markup languages 19
  See also object format rules *and* text format rules
format tags
  in conversion tables 473, 475, 476
  in text format rules 132, 144
formats
  creating automatically in templates 89
  storing in templates 103

G

general rules 111-116
  avoiding ambiguous 112, 114
  child elements in 112
  correspondence to markup languages 228
  default 115
  for empty elements 113
  grouping elements in 113
  restrictions on tables 114
  specifying text in 113
  syntax of 112-114
  table formats and 191
generate book (rule) 403
graphics
  comparison with markup languages 20
  defining elements for 99-102
  nesting in conversion tables 483
  object format rules for 191
  promoting in conversion tables 481
graphics and equations, default translation to and from markup languages
  anchored frame properties 302-304

creating graphic files on export 306
element and attribute structure 302
entity and file attributes 302
exporting entity declarations 305
graphic properties 304
on export to markup languages 300-305
on import from markup languages 307
text of default declarations 300
graphics and equations, modifying translation to and from markup languages
changing format of files 316
changing name of files 314
changing size of graphics 319
exporting elements 309
omitting elements and attributes 313
omitting graphic properties 312
renaming attributes for properties 311
renaming elements 309
representing structure of equations 311
specifying data content notation 313
specifying entity names 318
graphics, read/write rules for 358
anchored frame 365
entity name is 383
export dpi 387
export to file 389
facet 394
fm property 399
is fm graphic element 422
is fm property 424
is fm property value 426
is fm value 439
notation is 445
specify size in 456
value 464
value is 399

H

hidden and read-only attributes
specifying in definitions 174
HTML
conversion macros 48
elements mapped from FrameMaker elements 47
export 43
mapping for export 46

I

IDReference attributes
comparison with markup languages 21
defining 179
using for cross-references 176
if clauses, in format rules 134, 188
implied value is (rule) 406
imported graphic files. *See* graphics
importing element definitions 104
importing from markup languages. *See* markup languages, translation to and from
include dtd (rule) 408
INCLUDE keyword, in rules documents 214
include sgml declaration (rule) 409
inclusions
correspondence to SGML 233
in element definitions 117
when used with exclusions 117
indentation settings 147
inheritance of formatting information 127-131
in tables or footnotes 130
within books 131
initial conversion tables 473
initial structure pattern, for Rubi groups 123
initial structure pattern, for tables 121-122
insert table part element (rule) 410

is fm attribute (rule) 414
is fm char (rule) 416
is fm cross-reference element (rule) 418
is fm element (rule) 419
is fm equation element (rule) 420
is fm footnote element (rule) 421
is fm graphic element (rule) 422
is fm marker element (rule) 423
is fm property (rule) 424
is fm property value (rule) 426
is fm reference element (rule) 430
is fm rubi element (rule) 432
is fm rubi group element (rule) 432
is fm system variable element (rule) 433
is fm table element (rule) 435
is fm table part element (rule) 436
is fm value (rule) 439
is fm variable (rule) 441
is processing instruction (rule) 441
ISO Latin-1 character set 501-507
ISO public entities 509-515
  declarations and rules 514-515
  default character formats 513
  default variable definitions 513
  entity declaration files 511
  entity read/write rules files 511
  format of entity rules 512

K

keyboard shortcuts for an EDD 102

L

{last} sibling indicator 135, 189
last format rules 156-158
  how applied 157
  when to use 162
  with autonumbers 157
level rules (for formatting) 138-140
  ancestor tags in 139
  counts using current element 139
  order of clauses in 138
limits on formatting values 165
line break (rule) 443
line breaks and SGML record ends 233
log files
  generating 71
  hypertext links in 72
  limiting length of 70
  messages in 71
  See also errors

M

marker text is (rule) 444
markers
  comparison with SGML 23
  defining elements for 99-102
  list of predefined types 193
  object format rules for 192
markers, default translation to and from markup languages
  on export to markup languages 336
  on import from markup languages 336
markers, modifying translation to and from markup languages
  discarding nonelement markers 339
  identifying markers with attributes 338
  translating elements as markers 337
  writing marker text as content 337
markers, read/write rules for 359
  drop 373
  external data entity reference 391
  fm marker 398

fm property 399
is fm marker element 423
is fm property 424
is fm property value 426
is fm value 439
is processing instruction 441
marker text is 444
processing instruction 451
value 464
value is 399
markup language documents, read/write rules for 361
external dtd 392
include dtd 408
write structured document instance only 466
markup languages
comparison with FrameMaker 13-??
features not in FrameMaker 22
markup languages, translation to and from
books 347-353
cross-references 321-327
EDD content rules 118
elements and attributes 227-246
entities and PIs 247-269
example of 203-207
graphics and equations 297-320
markers 335-339
overview of 201-207
Rubi groups 237-238
tables 271-295
UniqueID values 179
variables 329-334
See also read/write rules *and specific element types*
{middle} sibling indicator 135, 189

N

nested format rules 140
notation is (rule) 445
{notfirst} sibling indicator 135, 189
{notlast} sibling indicator 135, 189
Numbering properties (text formatting) 153
numeric attributes 172
specifying a range for 175
values allowed in 173

O

object format rules 185-198
all-contexts rules in 187
context rules in 188-190
debugging 197-198
for cross-references 194
for equations 194
for graphics 191
for markers 192
for system variables 195
for tables 190
format rule overrides and 187
overview of 186
seeing which rules apply 197
sibling indicators in 189
translation to markup languages 185
object type identifiers, in conversion tables 476
{only} sibling indicator 135, 189
operators with attributes, for formatting 137
output book processing instructions (rule) 447

P

Pagination properties (text formatting) 152
paragraph formats
applying to particular contexts 144
building table structure from 484

finding errors in 166
how inherited from ancestors 127-131
setting a base element format 132
paragraphs
as a prefix or suffix 159, 161
formatting elements as 143
parentheses
in conversion tables 478
in general rules 113
PIs. *See* processing instructions
plus sign (+)
in conversion tables 478
in general rules 112
prefix rules 158-162
attributes in 162
defining for paragraphs only 159
defining for ranges and paragraphs 161
defining for text ranges only 159
how applied 158
when to use 162
preserve fm element definition (rule) 448, 449
processing instruction (rule) 451
processing instructions (PIs), default translation to and from markup languages
for condition settings 342
for conditional text 342
for marking book or document 348
on export to markup languages 248
on import from markup languages 256
processing instructions (PIs), default translation to and from SGML
PI entities 256
processing instructions (PIs), modifying translation to and from markup languages
discarding unknown instructions 268
suppressing creation of in books 353
processing instructions (PIs), read/write rules for 360
drop 373
fm marker 398
is processing instruction 441
output book processing instructions 447
processing instruction 451
use processing instructions 403
PROMOTE keyword 481, 482
proportional width resolution is (rule) 452
public identifiers 61
put element (rule) 403

Q

qualifiers, in conversion tables 471, 475, 480
question mark (?)
in conversion tables 478
in general rules 112
quotation marks ("), in attribute values 479

R

read/write rules
case conventions for 212
checking correctness of 215
commands for working with 215
comments in 214
constants in 213
documents for 42, 55, 209
for CALS tables 493-495
for creating an EDD from a DTD 77
including files with 64, 214
order of rules 210
overview of developing 35-38
reserved element names in 214
strings in 212
summary of 355-363

syntax for 211
uses for 201
variables in 213
See also specific rules and categories of rules
reader (rule) 453
read-only attributes
creating with read/write rules 244
reformat as plain text (rule) 454
reformat using target document catalogs (rule) 455
relative values, in format rules 128
retain source document formatting (rule) 456
Rubi groups
converting markup language elements to 237
defining elements for 97-98
initial structure pattern for 123
Rubi groups, read/write rules for
is fm rubi element 432
is fm rubi group element 432

S

sample documents
for testing an application 33-35
that come with FrameMaker 107
SDATA entities
default translation of internal 252
translating as characters 260-262
translating as reference elements 263
translating as text insets 262
translating as variables 259
translating entity references 258
SGML
comparison with FrameMaker ??-24
features not in FrameMaker 23
optional unsupported features 24, 500
SGML declarations 41
default for FrameMaker 497-499
for a DTD created from an EDD 107
specifying location of 57
SGML documents, read/write rules for
include sgml declaration 409
SGML parser
applying to a DTD 107
concrete syntax variants 499
SGML, translation to and from
batch export (UNIX) 519
batch import (UNIX) 517-519
sibling indicators
in format rules 135, 189
used with TEXT keyword 136, 189
spacing settings
for lines 148
for paragraphs 148
for words 154
line spacing and font sizes 149
specify size in (rule) 456
start new row (rule) 458
start vertical straddle (rule) 460
string attributes 172
structure, adding to documents. *See* conversion tables
structure API clients
specifying location of 66
uses for 202
structure applications
defining 53
location of files in 49
overview of 9-12
pieces of 41-43
process of developing 30-40
scenarios for 7
setting in an EDD 89

structure rules 109-124
debugging 123
overview of 110
Structure View
attributes in 170
cross-references in 177
invalid contents in 110
subrules, for format rules 140
suffix rules 158-162
attributes in 162
defining for paragraphs only 159
defining for ranges and paragraphs 161
defining for text ranges only 159
how applied 158
when to use 162
system variables
defining elements for 99-102
finding errors in definitions of 197
in conversion tables 477
list of predefined variables 196
object format rules for 195
system variables, default translation to and from markup languages
entities for nonelement variables 330
on import from markup languages 331
system variables, default translation to and from markup langugaes
on export to markup languages 330
system variables, modifying translation to and from markup languages
discarding variables 334
renaming or changing entity types 332
translating as variable elements 333
translating SDATA entities 259
translating to markup language text 333

T

tab stop settings 149
Table Cell properties (text formatting) 155
table ruling style is (rule) 461
tables
building structure from format tags 484
comparison with markup languages 21
default general rules for 115
default initial structure for 122
defining elements for 93-97
finding errors in formats of 197
formatting cells in 155
general rules for 111-116, 191
inheritance of text formats in 130
initial structure pattern for 121-122
nesting in conversion tables 483
object format rules for 190
paragraph formats for 132
promoting in conversion tables 481
restrictions on general rules 114
tables, default translation to and from markup languages
on export to markup languages 275
on import from markup languages 272-275
tables, modifying translation to and from markup languages
creating parts without content 287-289
creating tables inside tables 295
creating vertical straddles 290-293
exporting widths proportionally 290
formatting as boxed paragraphs 293
formatting with CALS attributes 282
identifying colspecs and spanspecs 283
omitting representation of parts 285-287
renaming table parts 279
representing properties as attributes 280

representing properties implicitly 281
rotating tables on a page 295
specifying columns for cells 284
specifying location of rows or cells 283
specifying ruling style for tables 289
table cell paragraph properties 279
table format properties 276-277
table straddle properties 278
tables, read/write rules for 361
end vertical straddle 379
fm property 399
insert table part element 410
is fm property 424
is fm property value 426
is fm table element 435
is fm table part element 436
is fm value 439
proportional width resolution is 452
start new row 458
start vertical straddle 460
table ruling style is 461
use proportional widths 463
value 464
value is 399
templates 42
creating an Element Catalog in 103-105
creating formats automatically in 89
specifying location of 57
storing formats in 103
text format rules 125-167
all-contexts rules in 133
context labels in 142
context rules in 134-138
debugging 166
element paragraph format in 132
first and last format rules 156-158
formatting specifications in 146-155
how inherited from ancestors 127-131
level rules in 138-140
limits on values in 165
multiple format rules 141
nested format rules 140
no additional formatting 145
overview of 126
paragraph formatting with 143
seeing which rules apply 167
sibling indicators in 135
text range formatting with 144
translation to XML 125
text insets, read/write rules for 362
entity 381
reformat as plain text 454
reformat using target document catalogs 455
retain source document formatting 456
TEXT keyword
in general rules 113
used with sibling indicators 136, 189
text ranges
as a prefix or suffix 159, 161
formatting elements as 144
text, read/write rules for
character map 369
entity 381
is fm char 416
line break 443
TEXTONLY keyword
changing SGML declared content 238
in general rules 113
Type 11 markers
discarding constructs using 268

U

UniqueID attributes
- comparison with markup languages 21
- defining 178
- translating IDs to markup languages 179
- using for cross-references 176
- values provided by end users 178
- values provided by system 179

untagged formatted text, wrapping 483
unwrap (rule) 461
use processing instructions (rule) 403
use proportional widths (rule) 463
user variables
- in container elements 197
- in conversion tables 477

user variables, default translation to and from markup languages
- entities for variables 330
- on export to markup languages 330
- on import from markup languages 331

user variables, modifying translation to and from markup languages
- discarding variables 334
- renaming or changing entity types 332

user variables, modifying translation to and from SGML
- translating SDATA entities 259

Using attibutes in a CALS table model 487

V

validity at highest level, specifying 116
value is (rule) 399
value (rule) 464
variables, read/write rules for 362
- drop 373
- entity 381
- fm element unwrap 397
- fm variable 402
- is fm system variable element 433
- is fm variable 441

variables. *See* system variables *or* user variables
version numbers in EDDs 88
vertical bar (|)
- in conversion tables 478
- in format rules 135, 188
- in general rules 112

W

wildcard characters
- in conversion tables 475
- in format rules 135, 188

wrapping with conversion tables
- document objects 476
- elements 477
- sequences of elements or paragraphs 478
- untagged formatted text 483

write structured document instance only (rule) 466
writer (rule) 467

X

XML, translation to and from
- conditional text 341-345